Universiteit Leiden

Master Computer Science

An approach to describing the semantics of Hedy, a gradual programming language for education

Name:	Julia Bolt
Student ID:	s1783769
Date:	[29/06/2022]
Specialisation:	Computer Science and Education
1st supervisor:	Dr. Henning Basold
2nd supervisor:	Dr. ir. Felienne Hermans

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Hedy is a gradual programming language for education. It consists of 18 levels, each supporting new commands or requiring code that satisfies new syntax rules, to gradually learn programming in Python. The semantics of a language describe the behaviour of programs in that language or the language as a whole, semantics give meaning to syntax. Formally describing and implementing the semantics of Hedy helps to improve it, by finding unwanted or illogical behaviours. Dafny can be used to implement semantics and compile it to other programming languages, for a reference implementation. Having a reference implementation of how this open source programming language should work is helpful for the people still developing Hedy and might encover issues.
Keywords: Big-Step Operational Semantics, Hedy, programming languages for novices, education, Dafny, software verification, Python

Contents

1 Introduction **1**

2 Hedy **2**
2.1 Why gradual? . . . 2
2.2 Python based . . . 2
2.3 Design Principles . . . 3
2.4 User Interface . . . 3
2.5 Changes per level . . . 4
2.6 Extensible and Growable Languages Outside of Education . . . 5
2.7 Programming Languages for Novices . . . 5

3 Semantics and Mathematical Preliminaries **7**
3.1 Syntax versus Semantics . . . 7
3.2 Denotational and Operational Semantics . . . 7
3.2.1 Small-Step and Big-Step Operational Semantics . . . 7
3.3 Set Theory . . . 8
3.3.1 Partial maps . . . 8
3.3.2 Coproducts . . . 9
3.4 Type Theory . . . 10

4 Dafny **11**

5 Description of the Syntax and Semantics of Hedy **12**
5.1 Informal Syntax and Semantics per level . . . 12
5.2 Big-Step Operational Semantics per level . . . 14
5.3 Implementation of Semantics of Hedy in Dafny . . . 32

6 Uncovered issues in Hedy **33**

7 Conclusions and discussion **34**
7.1 Future work . . . 34
7.2 Limitations . . . 35
7.3 Conclusion . . . 35

References **39**

Appendix **40**
A: Dafny implementation of level 1 . . . 40
B: Dafny implementation of level 2 . . . 42
C: Dafny implementation of level 3 . . . 46
D: Dafny implementation of level 4 . . . 51
E: Dafny implementation of level 5 . . . 56

1 Introduction

Hedy is a gradual programming language for education [Her20]. It consists of several levels, starting out very simple with only a few commands and as little syntax rules as possible. Every next level supports new commands or requires a code that satisfies new syntax rules. After the final level, Hedy code actually corresponds with Python code. In the case of the Hedy programming language, every level has its own syntax and semantics.

Just like natural languages, programming languages consist of syntax and semantics. Syntax tells you how to write something (spelling rules) and semantics give meaning to this syntax. In the case of natural languages, this can be described and will sometimes show that not everything is completely logical, and the language might change overtime, so the syntax and semantics might also change. In the case of a programming language, syntax tells us how to write things like where to put a colon, for example. Semantics is what gives meaning to the syntax, it explains the behaviour of a language. Different from a natural language, a programming language did not arise at some point, and might evolve from there, but programming languages are created, (usually) with a plan behind it. New programming languages can make programming easier in general, or for a specific domain. For programming languages, we can do more than just describe the syntax and semantics, we can also make some changes to make the language more intuitive. Designing and implementing a programming language is often a work-in-progress, so the description of syntax and semantics might also be used to check the developers, to see if changes made to a language are consistent with the syntax and semantics. When designing a new programming language, it is useful to have a detailed description of how it should work. Especially with multiple people working on it at the same time.

Dafny is a programming language for software verification [Mica, Micb]. It compiles to other programming languages, like Java, C# and Go. Implementing semantics in Dafny and compiling it to for example C#, to make a reference implementation, could help finding issues in the implementation of Hedy. Thus, it might be interesting to formally describe the semantics of Hedy and based on that, make a reference implementation in Dafny. The goal would be to improve Hedy, by finding issues. The research question that follows is:

How can we uncover issues in the implementation of the Hedy programming language, by providing a formal description of its semantics and a formalised implementation?

This chapter contains the introduction; Section 2 includes background information and related work on Hedy; Section 3 includes explanations on used mathematics and on semantics; Section 4 discusses background information and related work on Dafny; Section 5 describes the semantics of Hedy; Section 6 describes the uncovered issues in Hedy; Section 7 concludes.

I would like to thank my supervisors, dr. Henning Basold and dr.ir. Felienne Hermans, both from Leiden Institute of Advanced Computer Science [lia], for their guidance during the research project, their feedback and encouragement, also during the writing process. I have learned a lot and I am looking forward to the teacher's programme, bringing things I've learned into practice and combining computer science with education in the future, possibly as a teacher and possibly as a researcher.

2 Hedy

Hedy is a gradual programming language, designed for education purposes [Herc, Herb, Her20]. The target audience consists of novices from the age of about 11. [Her20] It consists of multiple levels and starts very simple with only a few possible commands and as little syntax rules as possible. New syntax rules and new programming concepts are introduced in the higher levels. In the highest levels, Hedy code is the same as Python code. The language works in the browser and is available in many (natural) languages, which should make the barrier to get started low. More content and translations are added regularly. The English and Dutch versions have the biggest amount of levels and content in the form of examples and explanations. The language is still in development and it is open source, with many contributors [Herb]. The question might arise, why a gradual programming language? And why is it based on Python? These questions will be addressed in this section. The design principles and changes per level will be listed and the user interface will be shown. Hedy will also be compared to other programming languages later in this section.

2.1 Why gradual?

The syntax of programming languages is one of the aspects of programming that novices commonly struggle with: remembering the appropriate commands to use and arranging them into a working program. [Her20] Hedy is presented as a new way of teaching the syntax of a programming language to novices, inspired by educational methods by which punctuation is taught to children. Programming is frequently associated with STEM (Science, Technology, Engineering, and Mathematics) [SKB+13, WBH+16], however learning a programming language has many similarities to learning a natural language, since learners must learn about both semantics and syntax. It has been suggested that using instructional tactics similar to those used in natural language teaching might improve programming education [HA17, Por18]. Inspired by these tactics for natural language education, Hedy has been designed as a gradual language for programming education.

2.2 Python based

Python has been compared to other programming languages, to understand the effect of different languages on study success of learners. The research outcomes suggest that Python has benefits in teaching. Comparing a course in Python to one in C, Wainer and Xavier found that students were less likely to fail and scored higher on the exam when using Python [WX18]. Also when starting out with Python and learning Java later, no disadvantages have been found in research. [MPS06] Thus, learning a simpler language does not seem to interfere with learning a more complicated language later. Python might even serve as a good preparation for subsequent C++ courses. [EP10, EPM09] According to research comparing Python with Java for programming education, Python makes basic concepts easier to introduce in quite a few cases, because of less *syntactic noise* and less *conceptual noise*. [PDP19] A short list has been proposed, of syntax- and semantics-related desiderata for a beginner language (which neither Java nor Python completely answer). *Desideratum 1: The first steps in a language (e.g., console print, variable assignment) should be painless and minimize both the syntactic noise and the conceptual noise. [PDP19]* This fits with the design principles of Hedy.

2.3 Design Principles

According to the creator of this programming language, Felienne Hermans, Hedy follows these design principles. [Her20]

1. Concepts are offered at least three times in different forms
2. The initial offering of a concept is the simplest form possible
3. Only one aspect of a concept changes at a time
4. Adding syntactic elements like brackets and colons is deferred to the latest moment possible
5. Learning new forms is interleaved between concepts as much as possible
6. At every level it is possible to create simple but meaningful programs

2.4 User Interface

Figure 1 shows the current Hedy implementation. On the left, there's an editor for entering code, and on the right, there's a field for output. Each level also includes buttons that allow you to try the commands you've learned so far. Videos with explanations and written assignments are also available from within the user interface for each level.

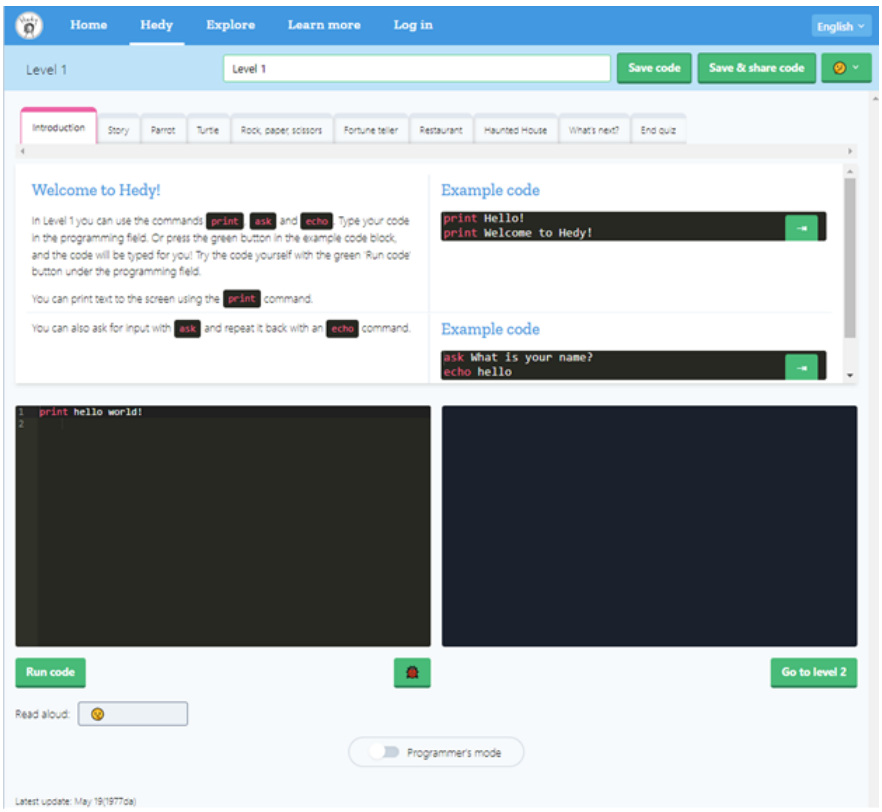

Figure 1: Level 1 of the Hedy user interface in English [Herc]

2.5 Changes per level

Currently, Hedy consists of 18 levels and the changes made per level are given here for a short overview. There are plans to increase the amount of levels, to make some steps/gaps between levels smaller.

- **Level 1** supports **print**, **ask**, **echo**, **turn** and **forward**.
- **Level 2** adds **is** for assignment and **sleep**. The command **echo** is not supported anymore, and **turn** cannot be used with **left** or **right** anymore (only with nothing or an integer).
- **Level 3** adds lists, with commands **at random**, **add to**, **remove from**.
- **Level 4** adds quotes around text used with **print** and **ask**.
- **Level 5** adds **if** and **if else**, where the if-condition can be equality comparison with **is** or checking if something is an element of a list with **in**. (Note that **is** can now be assignment or comparison!)
- **Level 6** adds **=** for assignment or comparison, and calculations with integers are now possible.
- **Level 7** adds **repeat**.
- **Level 8** adds indentation.
- **Level 9** adds nesting.
- **Level 10** adds **for ... in ...**
- **Level 11** adds **for ... in range ... to ...**. Now, **repeat** is not possible anymore.
- **Level 12** adds floats and quotes around all text (so now also when used with assignment, lists and **if**).
- **Level 13** adds **and** and **or**.
- **Level 14** adds comparisons: $<$, $>$, $<=$, $>=$, $==$, $!=$. (Note that now **is**, $=$ and $==$ can be used for comparison and **is** and $=$ for assignment!)
- **Level 15** adds **while**.
- **Level 16** adds [ ] for lists. Using **print** and **ask** with a list is now possible.
- **Level 17** adds **elif** and a colon before every indentation (so behind **if**, **for** and **while**).
- **Level 18** adds open and close paren to **print** and **range**, the syntax becomes **print(...)** and **range(...)**

2.6 Extensible and Growable Languages Outside of Education

Outside of programming education, languages have been proposed that change over time. For example, Fortress [ACN+09] is a growable language and Lusth et al. [LKT09] proposed using reflection and overloading to grow and shrink languages. These languages were not proposed for education, different from Hedy, but other reasons can be found to prefer a programming language that changes. Fortress was designed to accommodate the changing needs of its users, and the proposal by Lusth et al. could be useful for example to comply with specific coding style guidelines. The purpose of Hedy is to teach programming.

2.7 Programming Languages for Novices

Hedy is the realisation of a new approach to programming education. Hedy is not the first programming meant for education though. Three different approaches in programming languages for novices can be identified in prior research [B+94, Her20]: Mini-languages, sub-languages and the incremental approach. Hermans gives examples of languages for all three approaches and explains how these approaches are different from the gradual approach that Hedy uses. Mastering a mini-language can be a goal in itself, leading to algorithmic thinking, but Hedy is designed to help novices learn Python eventually. Sub-languages use only a set of commands from a larger programming language, usually one that is used in practice. Initially, the goal of sub-languages was not to gradually expand them, like Hedy, but simply to select a subset to teach. The incremental approach teaches a small subset of a programming language, with each subset introducing new programming constructs. Other forms of incremental teaching used subsets that were specifically not organized in a hierarchy, where the "higher level" contained the "lower level", like in Hedy, but instead separated the language into overlapping languages, similar to how chapters in a textbook would. [Her20]

Kelleher and Pausch give a more extensive overview of different kinds of programming languages for novices. [KP05] Figure 2 shows the novice programming systems and languages taxonomy according to their study. The systems in this taxonomy are divided into two categories: those that seek to teach programming for its own sake and those that attempt to facilitate the use of programming to achieve a specific purpose. The taxonomy is arranged first by the system aims, either teaching or utilizing programming, and then by the major feature of programming that the system seeks to simplify. Each system appears just once in the taxonomy. However, many of the taxonomy's systems are based on earlier systems' principles. As a result, a system that was affected by natural language programming may not be classed alongside other natural language systems if the system's major contribution was not to enable natural language programming.

The idea for a language similar to Hedy has been descibed before [Shn77]. In the paper, no implementation was given in a language or lesson series though. Some other languages extend over time exist and share philosophical principles with Hedy. For example, Cazzola and Olivares gradually build up to JavaScript [CO16]. Cupi2 is Java-based, in which students solve increasingly more complicated problems with partly generated programs [VJV13]. DrScheme set of languages, each a larger subset of Scheme [FFFK04]. However, a language of which the syntax gradually changes, rather than being extended, has not been described or implemented before.

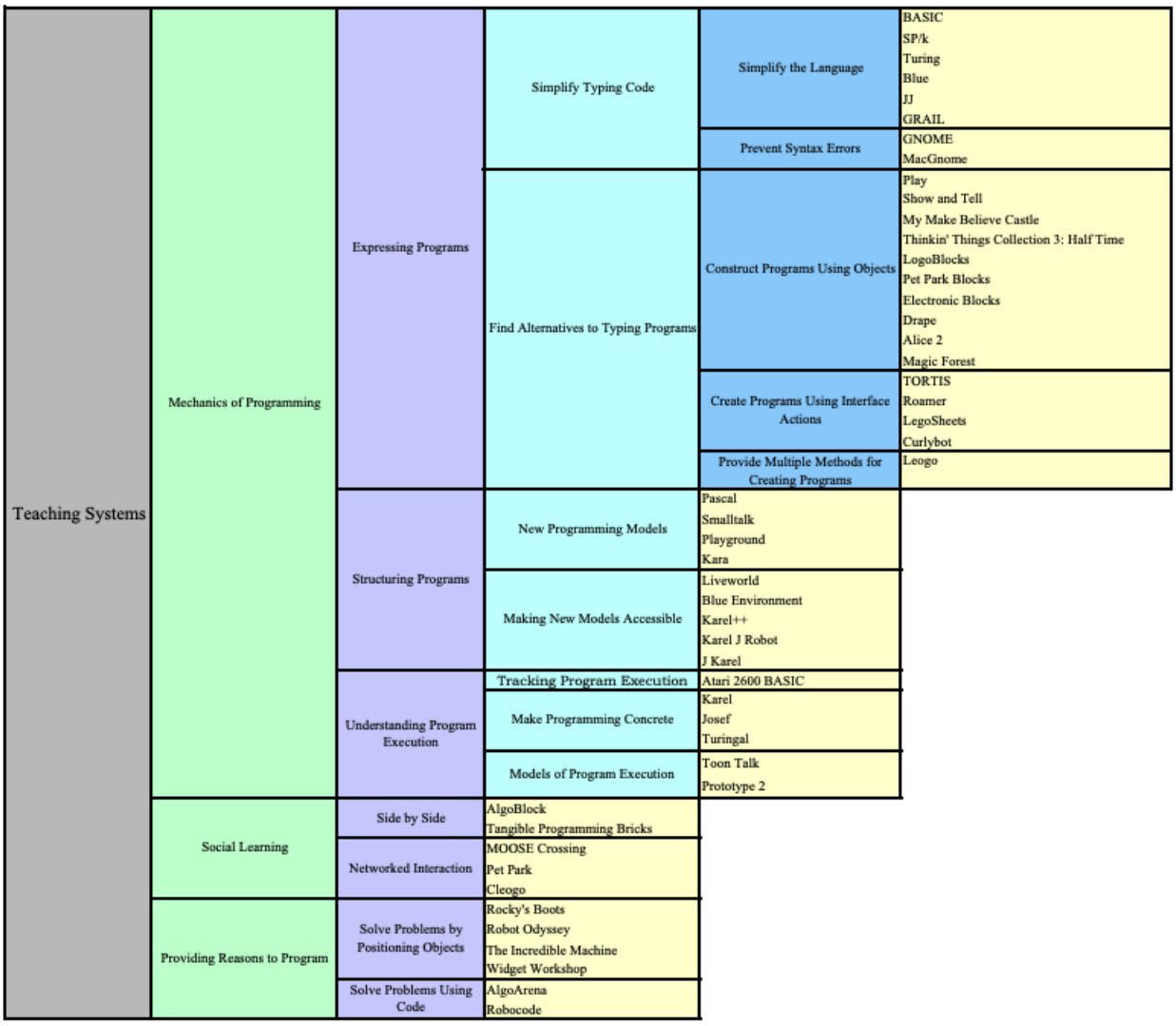

Figure 2: Novice Programming Systems and Languages Taxonomy. [KP05]

3 Semantics and Mathematical Preliminaries

Before diving into the semantics of Hedy specifically, a brief introduction into syntax, semantics and category theory is given in this section. First, the difference between syntax and semantics will be explained, then the different methods for describing the semantics of a programming language is discussed. Then some set theory and type theory will be introduced.

3.1 Syntax versus Semantics

A computer language's syntax is a collection of rules that specify the symbol combinations that are regarded correctly formed statements or expressions in that language. Syntax relates to the structure of the code, but semantics, on the other hand, refers to the content or meaning [BM17, Win93]. The syntax of a language specifies the structure of a valid program, but it does not tell you anything about the program's content or the outcomes of running it. Semantics deals with the meaning provided to a combination of symbols (either formal or hard-coded in a reference implementation). Not all programs that are syntactically correct are semantically correct. In natural language, we also speak of syntax and semantics, and it works the same there. It may not be possible to assign a meaning to a grammatically correct sentence or the sentence may be false, for example:

- "Colorless blue ideas sleep furiously." is grammatically well formed but has no generally accepted meaning.
- "Paul is a married bachelor." is grammatically well formed but expresses a meaning that cannot be true.

3.2 Denotational and Operational Semantics

The two main techniques to describing the semantics of a programming language are denotational semantics and operational semantics [BM17, Win93]. In denotational semantics, the meaning of a program is expressed in terms of mathematical objects that represent expressions and their manipulations. Operational semantics express the meaning of a program through logical statements about its execution. Denotational semantics tend to be more powerful, as properties of the mathematical objects used can aid in proving properties of the programs, but are harder to write down. Operational semantics tend to be easier to write down and are more closely related to the execution of programs. As a result, they are more valuable as a basis for language implementations. This thesis will be using operational semantics.

3.2.1 Small-Step and Big-Step Operational Semantics

There are two methods for constructing proofs in operational semantics: big-step and small-step. Big-step semantics describe how to break down a calculation into smaller sub-calculations and how the results should be combined. Small-step semantics simply provide rules for how all of a program's components should develop. The meaning of a program is then formed by applying the proper rules repeatedly. In this thesis, big-step semantics will be discussed exclusively.

3.3 Set Theory

Sets and types are not the same thing, but they are similar and can likewise be used as a foundation of mathematics. First, we give an explanation of the way set theory is defined, to show how this differs from type theory. A set theory is typically built on top of a logic. A logic can be defined by giving a number of inference rules. The inference rule $(Rule)$ says that from the premises $p_1, ..., p_m$ we can deduce the conclusions $c_1, ..., c_n$.

$$\frac{p_1 ... p_m}{c_1 ... c_n} \ (Rule)$$

An example of such an inference rule in propositional logic, is modus ponens, which says that if A implies B and we have A, then we can deduce B.

$$\frac{A \Longrightarrow B \ A}{B} \ (MP)$$

Another example on an inference rule is the law of the excluded middle, which states that without any premises, we can conclude that A is true or A is not true.

$$\frac{}{A \vee \neg A} \ (LEM)$$

After we define our logic by stating inference rules, we define our set theory by taking a number of axioms, these are statements we assume without proof. A formal proof is now an ordered list of statements such that each statement can be deduced from earlier statements or axioms by some inference rule.

3.3.1 Partial maps

A partial function f from a set X to a set Y is a function from a subset S of X to Y. f is said to be total if $S = X$, that is, if f is defined on every element in X. [Win93] More technically, a partial function, is a binary relation over two sets that maps every element of the first set to *at most* one element of the second set, it is thus a functional binary relation. It generalises the definition of a (total) function by not requiring that each element of the first set be mapped to *exactly* one element of the second. When the exact domain of definition is unknown or impossible to establish, a partial function is frequently utilised. This is true in calculus, where the quotient of two functions, for example, is a partial function whose domain of definition cannot contain the denominator's zeros. A partial map f from a set X to a set Y can be written as:

$$f : X \rightharpoonup Y$$

In this thesis, the partial map will be useful for defining memory Σ as a map from variable locations/names to their values. We need a partial map for this, because not every possible variable name/location will be used and map to a value. Another important notion for defining the memory is that the values are not always of the same type and thus not elements of the same set. This means that the memory is actually not one partial map, but a set of multiple partial maps. The notation used for this are rectangular brackets:

$$\Sigma = [X \rightharpoonup Y]$$

The domain X will be all possible variable names/locations, but the codomain Y is more difficult, since it should be multiple different sets, depending on the type of the value. For this, we can use a coproduct of sets.

3.3.2 Coproducts

In order to understand what the *coproduct* of sets is, it might be useful to first introduce the *product* of sets. The *product* of two sets X and Y, is the set of all pairs with the first value taken from X and the second taken from Y. This can be defined as:

$$X \times Y = \{(x, y) \mid x \in X, y \in Y\}$$

The *coproduct* of sets is the same as their disjoint union. If sets X and Y are disjoint, so they share no elements, the disjoint union $X \coprod Y$ is the same as the (regular) union:

$$X \cup Y = \{x \mid x \in X \text{ or } x \in Y\}$$

If the original sets are *not* disjoint, we need to give some index to each element denoting the originating set. Then, elements that are present in two sets will also appear twice in the disjoint union, with their own label or index. This can not only be done for two sets, but also for more sets of course. The disjoint union, or coproduct of sets, can be defined as:

$$\coprod_{a \in A} X_a = \{(a, x) \mid a \in A, x \in X_a\}$$

For example,

$$\begin{aligned} X_0 &= \{4, 5, 6\} \\ X_1 &= \{5, 7\} \\ X_2 &= \{6, 7\} \\ \coprod_{a \in \{0,1,2\}} X_a &= \{(0,4), (0,5), (0,6), (1,5), (1,7), (2,6), (2,7)\} \end{aligned}$$

In Figure 3, a visualisation can be found of the disjoint union or coproduct of sets.
We can use the coproduct in the definition of the memory. As explained before, the memory Σ can be defined as partial maps from a set of variable names/locations to their values. Since the values can be of multiple types, they are elements of different sets, which form the codomain. The codomain can be defined as the coproduct or disjoint union of these sets. If we only have integers and strings as types, like we have in level 2 of Hedy, the codomain is $\mathbb{Z} \coprod A^*$ for some alphabet A. This gives us a definition of the memory Σ of the form:

$$\Sigma = [X \rightharpoonup \coprod Y]$$

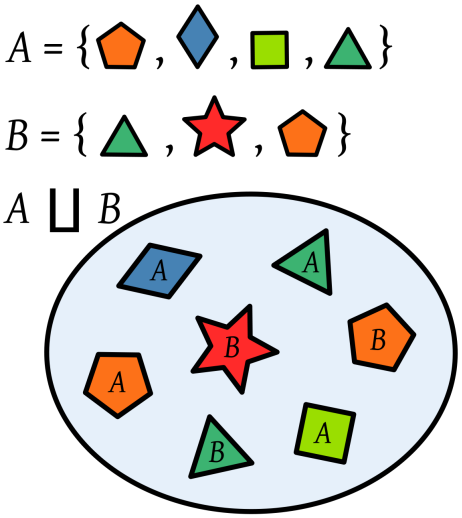

Figure 3: The disjoint union $A \coprod B$ of the sets A and B is the set formed from the elements of A and B labelled (indexed) with the name of the set from which they come. So, an element belonging to both A and B appears twice in the disjoint union, with two different labels. [Wik22]

3.4 Type Theory

In contrast to set theory, type theory [Pie02, Geu08, Pie04, Tho91] is not built on top of logic but is instead constructed from the ground up using inference rules. A type theory is a formal system in which every "term" has a "type". A "type" in type theory has a role similar to a "type" in a programming language: it dictates the operations that can be performed on a term and, for variables, the possible values it might be replaced with. We write $a : A$ if a is a term of the type A. In contrast to set theory, where elements can belong to several sets, a term usually only has one type. Note that this is the way that types work in programming languages, when we speak of the type `char` for characters like the term `'a'` and type `int` for integers like the term 123. Like Python, Hedy has a dynamic type system. We can use the following notation to be sure of the correctness of certain properties before execution.

$$\frac{v : T \in \Gamma}{\Gamma \vdash v : T}$$

Here, v is a variable of type T in the context Γ. We can use this notation for our description of Hedy level 3 and further.

4 Dafny

Dafny [Lei10] is an imperative and functional compiled language that enables formal specification through preconditions, postconditions, loop invariants, and loop variants, and compiles to other programming languages such as C# or Java. The language integrates elements from both the functional and imperative paradigms, with some object-oriented programming support. Features include generic classes, dynamic allocation, inductive datatypes, and implicit dynamic frames [SJP09], a variation of separation logic for reasoning about side effects. Dafny was created to provide a simple introduction to formal specification and verification, and it has been widely used in education. Dafny builds on the Boogie intermediate language which uses the Z3 automated theorem prover for discharging proof obligations [Z3g, dMB08]. Dafny is frequently included in software verification competitions (e.g. [LM10, KMS+11, BBD+12]). For other languages than Hedy, Dafny has also been used to implement the semantics. For example, for a course at Carnegie Mellon University on Automated Program Verification and Testing, lecture notes show how the language IMP can be implemented in Dafny. [Fre] Dafny is thus an interesting language for us to use, to create a reference implementation, by making an implementation based on the semantics of Hedy.

5 Description of the Syntax and Semantics of Hedy

The starting point for describing the semantics of Hedy, was to gain an overview of the possible commands in every level and the types to be used with every command. I have done this for all 17 levels of Hedy that existed at that time (now, level 18 is the final level). Apart from the commands and types per level, the file also contains examples of correct and incorrect Hedy programs. Because Hedy code is transpiled into Python, the semantics of an Hedy program is defined in terms of the semantics of the resulting Python code, or, in case of an error, the resulting HedyException. After completing this informal overview of the semantics of Hedy, a formal description could be made. For this thesis project, only the first 5 levels are formally described and implemented, to illustrate a possible approach.

5.1 Informal Syntax and Semantics per level

Here is only the first level given, the complete overview can be found on GitHub [Herb] in the file SEMANTICS.md. The changes per level are also listed in Section 2.

Level 1

Commands and types

Level 1 supports:

Command	Types
`print`	string
`ask`	string
`forward`	integer | empty
`turn`	'right' | 'left' | empty
`echo`	string | empty

Table 1: Commands and types supported by Hedy level 1

Correct Programs

Hedy	Python
`print x y`	`print('x y')`
`forward`	`t = turtle.Turtle()` `t.forward(50)`
`turn`	`t = turtle.Turtle()` `t.right(90)`
`turn left`	`t = turtle.Turtle()` `t.left(90)`
`turn right`	`t = turtle.Turtle()` `t.right(90)`
`ask x`	`answer = input('x')`
`ask x` `echo y`	`answer = input('x')` `print('y', answer)`

Table 2: Examples of correct programs in Hedy level 1 and the resulting Python code

Incorrect Programs

Hedy	**Exception**
`print`	Incomplete Exception
`ask`	Incomplete Exception
(`forward` | `turn`) text	Invalid Argument Type Exception
(not (`print` | `ask` | `forward` | `turn` | `echo`)) (any text)?	Invalid Exception
any valid line(s) of code not using `ask` `echo x`	 Lonely Echo
(space) (`print` | `ask` | `forward` | `turn` | `echo`) (any text)?	Invalid space

Table 3: Examples of incorrect programs in Hedy level 1 and the resulting HedyException

In the complete overview, there are three tables similar to Table 1, 2 and 3 for every level. This informal description of the semantics has been helpful for creating a formal description, and it is also useful as documentation for the developers of Hedy.

5.2 Big-Step Operational Semantics per level

Level 1

In level 1, there are only 5 possible commands, **print, ask, echo, turn, forward**. Those commands cannot all be used in the same way, as shown in the tables with commands and types per level in the informal description of the semantics, they need to be used with certain types. To make this formal, we make a grammar, showing what an element c of the set $Commands$ can consist of. In the grammar, we make sure the commands are used with the proper types and apart from listing the 5 possible commands, we add sequential composition, because a program can consist of more than one command. In the the following grammar, s is a string, so $s \in A^+$ for some alphabet A and n is an integer, so $n \in \mathbb{Z}$. For sequential composition, we use a semicolon, for a better readability in the following grammars and derivation rules. In Hedy code, the semicolon is not used, instead there is a newline. Rectangle brackets mean [optional]. The epsilon symbol ϵ means 'empty', this is needed because the commands **turn** and **forward** can be used without anything else, the turtle will turn with a default amount of degrees and move forward with a default amount of steps.

$$
\begin{aligned}
c &::= \textbf{print}\, s \mid \textbf{ask}\, s \mid \textbf{echo}\, [s] \mid \textbf{turn}\, d \mid \textbf{forward}\, p \mid c; c \\
d &::= \epsilon \mid \textbf{left} \mid \textbf{right} \\
p &::= \epsilon \mid n
\end{aligned}
$$

This grammar shows that commands like the following ones are not possible:

```
turn around
forward right
```

The grammar also shows that these commands are possible:

```
turn left
forward 50
```

Now that the grammar for level 1 is clear, we need to understand what the effect of these commands is. To do that, we need to establish an environment. We define that E_1 is an environment which includes certain functions. For the **print** command, we need a function that can print a given output, so it takes a non-empty output string and an environment and returns a changed environment:

$$print\colon A^+ \times E_1 \to E_1$$

For the command **ask**, we need to print a question, but since it is printed in a separate box, we cannot use the same function as the one we use for **print**. We also need to get an input from a user. This function should thus take a non-empty output string and an environment and return the input string and the changed environment:

$$ask\colon A^+ \times E_1 \to A^* \times E_1$$

The command **echo** prints something and then repeats the last answer given by the user after an **ask** command. So first, we need to check that there has been given an answer (after an

ask) that can be repeated, for this we have the function $echo\_possible$. If the answer exists and is not empty, add this answer to the string we want to print, and then print it (using the $print$ function we already have). The $echo\_possible$ function takes the given input and returns a boolean, determining whether it is not empty.

$$echo\_possible\colon A^* \to \mathbb{B}$$

The commands **turn** and **forward** make a 'turtle' move. They both take an integer to determine how far to move and thus both need a function that takes an integer and changes the environment:

$$turn\colon \mathbb{Z} \times E_1 \to E_1$$
$$forward\colon \mathbb{Z} \times E_1 \to E_1$$

The function, $draw$, is used for the turtle commands, if there is no turtle yet, this will be initialised. If there is a turtle already, since there were turtle commands used earlier in the program, $draw$ is not used again, the turtle only needs to be initialised once.

$$draw\colon E_1 \to E_1$$

In the following example, the functions that need to be used are first $draw$, then $turn$ and finally $forward$. We do not need to use $draw$ for the second command again, because we keep the same turtle moving and we do not want to add another turtle. This $draw$ function might seem unnecessary, but it is also what happens in the Python code that Hedy code gets translated into, to run.

```
turn left
forward 50
```

This Hedy code translates to the following Python code:

```
t = turtle.Turtle()
t.right(90)
t.forward(50)
```

The attentive readers will have noticed that the $turn$ and $forward$ functions both need an integer as input, but the command **turn** is used with **left** or **right** or nothing according to the grammar, and **forward** can also be used without any integer. We want our d, used with the command **turn**, to reduce to an $n \in \mathbb{Z}$, and our decision is to use a negative number for turning to the right and a positive number for turning to the left. This corresponds to the following rules.

$$\frac{}{\epsilon \Downarrow -90}\ (\epsilon_{turn}) \qquad \frac{}{\textbf{left} \Downarrow 90}\ (left_{turn}) \qquad \frac{}{\textbf{right} \Downarrow -90}\ (right_{turn})$$

The command **forward**, optionally uses an integer, but we always want to reduce that to an $n \in \mathbb{Z}$ too, whether the number is specifically given or not, so that we can always give a number to the $forward$ function. For this, we can use the following rules.

$$\frac{}{\epsilon \Downarrow 50}\ (\epsilon_{forward}) \qquad \frac{}{n \Downarrow n}\ (num)$$

These rules make sure that for example

```
turn
forward
```

corresponds with

```
turn right
forward 50
```

and the functions $turn$ and $forward$ will get -90 and 50 as input respectively.

A state in level 1 needs to include multiple elements. The environment, the answer that was given by the user after the most recent **ask** command and a boolean to indicate whether there is a turtle already or not. Thus, a state in level 1 is of the form:

$$S_1 = E_1 \times A^* \times \mathbb{B}$$

So for every command c, we want to have rules of the form: $\frac{}{< e, s, b, c > \Downarrow < e', s', b' >}$

Here, $e, e' \in E_1$, $s, s' \in A^+$, which is the last input/answer a user has given after an **ask** command, we use this for the **echo** command, and $b, b' \in \mathbb{B}$ to indicate whether there already is a turtle or not. 1 means there is a turtle, 0 means there is not. We are still using rectangle brackets to show something can be empty and a semicolon instead of a newline.

When the command c is of the form $\mathbf{print}\, x$, the function $print$ is used with string x and returns a new environment. The last input/answer given by the user does not change and the boolean to indicate whether there is a turtle also does not change. This gives the following rule. It says that the state we can find on the left of $\Downarrow$ evaluates to the state on the right of the $\Downarrow$ provided that the $print$ function with input (x, e) gives e' as output.

$$\frac{print(x, e) = e'}{< e, s, b, \mathbf{print}\, x > \Downarrow < e', s, b >}\ (print_{lvl1})$$

When the command c is of the form $\mathbf{ask}\, x$, the function ask is used to print the question x and the function returns the answer that the user gives. This input/answer is stored as a string, it may overwrite an older input/answer. In the following rule, the s' is the input the user gives and thus the string that the function ask returns, this is thus the string stored in the new state we can find on the right side of the $\Downarrow$.

$$\frac{ask(x, e) = (s', e')}{< e, s, b, \mathbf{ask}\, x > \Downarrow < e', s', b >}\ (ask_{lvl1})$$

When the command c is of the form $\mathbf{echo}\, [x]$, the function $echo\_possible$ is used to see if s, the variable in which the latest answer of the user should be stored, indeed contains something. It might be the case that there has not been asked a question and that s is still empty. In that case, we cannot repeat anything and thus the **echo** cannot be done. This means that only when $echo\_possible$ returns true, we can print. If it returns false, a HedyException should occur. In this description of the semantics and these rules, we do not explicitly show all the possible HedyExceptions. The exceptions correspond with a failure of evaluating Hedy code

using our rules. Thus, for **echo**, we only need a rule for the case that $echo\_possible$ returns true and we can actually print something repeating the answer given by user earlier.

$$\frac{echo\_possible(s) = \text{tt} \quad print(x + s, e) = e'}{< e, s, b, \mathbf{echo}\,[x] > \Downarrow < e', s, b >} \; (echo_{lvl1})$$

When the command c is of the form $\mathbf{turn}\,d$, and we already have a turtle, so our $b = 1$, we first need to make sure the d is evaluated to an integer n (using the rules $\epsilon_{turn}, left_{turn}, right_{turn}$ given earlier), and then we can use that integer for the function $turn$, as in the following rule.

$$\frac{d \Downarrow n \quad turn(n, e) = e'}{< e, s, 1, \mathbf{turn}\,d > \Downarrow < e', s, 1 >} \; (turn1_{lvl1})$$

When the command c is of the form $\mathbf{turn}\,d$, but we do not have a turtle yet, so our $b = 0$, we first need to initialise the turtle using $draw$. After $draw$, we get $b = 1$ and a new environment e'. The state that this evaluates to using the $turn1_{lvl1}$ rule we have just defined, is the same as the state we evaluate to here.

$$\frac{draw(e) = e' \quad < e', s, 1, \mathbf{turn}\,d > \Downarrow < e'', s, 1 >}{< e, s, 0, \mathbf{turn}\,d > \Downarrow < e'', s, 1 >} \; (turn0_{lvl1})$$

When the command c is of the form $\mathbf{forward}\,p$, we can use similar rules as for **turn**. First, we have the case that there is a turtle already, so $b = 1$. The p is evaluated to an integer n (using the rules $\epsilon_{forward}$ and num given earlier), and then we can use that integer for the function $forward$, as in the following rule.

$$\frac{p \Downarrow n \quad forward(n, e) = e'}{< e, s, 1, \mathbf{forward}\,p > \Downarrow < e', s, 1 >} \; (forward1_{lvl1})$$

When the command c is of the form $\mathbf{forward}\,p$, but now we have the case that there is no turtle yet, $b = 0$, we need to initialise the turtle first, using $draw$. After $draw$, we get $b = 1$ and a new environment e'. The state that this evaluated to using the $forward1_{lvl1}$ rule we just defined, is the same as the state we evaluate to here.

$$\frac{draw(e) = e' \quad < e', s, 1, \mathbf{forward}\,p > \Downarrow < e'', s, 1 >}{< e, s, 0, \mathbf{forward}\,p > \Downarrow < e'', s, 1 >} \; (forward0_{lvl1})$$

Last but not least, we need to define a rule for sequential composition. The different commands are not completely independent after all. If we have two commands c, c' after each other, the first command c changes the state and this state has to be the input state for the second command c', to again change the state and give us our final state. The input state with the first command can evaluate to a second state using the rules given so far. The output state is the input state for the second command and these can then again be evaluated using the rules given so far. The output state we get is the final one we evaluate to here. (But of course, one of these commands can again be an instance of sequential composition and this way we can

continue, evaluating a bigger program.)

$$\frac{< e, s, b, c > \Downarrow < e', s', b' > \qquad < e', s', b', c' > \Downarrow < e'', s'', b'' >}{< e, s, b, \mathsf{c\ ;\ c'} > \Downarrow < e'', s'', b'' >}\ (composition_{lvl1})$$

Using these rules, a derivation tree (or prooftree) can be made to show how a Hedy program can be evaluated. In the implementation of Hedy, exceptions are being used, which correspond with a failure of evaluating Hedy code using these rules. The same holds for the next levels. For example, a derivation tree for the following correct program:

```
turn
forward
```

would look like this:

$$\dfrac{draw(e) = e' \qquad \dfrac{\dfrac{}{\epsilon \Downarrow -90}\ (\epsilon_{turn}) \qquad turn(n, e) = e'}{< e, s, 1, \mathbf{turn} > \Downarrow < e', s, 1 >}\ (turn1_{lvl1})}{< e, s, 0, \mathbf{turn} > \Downarrow < e', s, 1 >}\ (turn0_{lvl1}) \qquad ***}{< e, s, 0, \mathbf{turn; forward} > \Downarrow < e'', s, 1 >}\ (composition_{lvl1})$$

The complete derivation tree does not fit on the page. Replace $***$ by the following subtree:

$$\frac{\dfrac{}{\epsilon \Downarrow 50}\ (\epsilon_{forward}) \qquad forward(n, e) = e'}{< e', s, 1, \mathbf{forward} > \Downarrow < e'', s, 1 >}\ (forward1_{lvl1})$$

Note that the function $draw$ is only used in the left subtree, when we are still evaluating the first command, $turn$. For the second command, $forward$, we do not need $draw$ anymore, which we know because we changed the $b = 0$ in our state to $b = 1$. Also note that the resulting state after evaluating $turn$ in the left subtree is the same as the initial state in the right subtree.

An example of a derivation tree for an incorrect Hedy program cannot be completed, for example, in the case there is only an **echo**, without **ask**:

$$\frac{echo\_possible(s) = \text{ff} \ldots}{< e, s, b, \mathbf{echo} > \Downarrow \ldots}\ (echo_{lvl1})$$

There is no rule for **echo** if $echo\_possible$ returns false, so we cannot get any further. Error! Another possible incorrect program would be when **turn** is used with an integer or text other than $left$ or $right$. Or **forward** with any text. For example, when the Hedy program looks like:

```
forward jump
```

it would give the following tree:

$$\dfrac{draw(e) = e' \qquad \dfrac{jump \Downarrow ... \qquad forward(..., e) = ...}{< e, s, 1, \textbf{forward}\, jump > \Downarrow ...}\ (forward1_{lvl1})}{< e, s, 0, \textbf{forward}\, jump > \Downarrow ...}\ (forward0_{lvl1})$$

Since 'jump' cannot be evaluated to an integer, the $forward$ function cannot be used and thus we cannot evaluate this program.

Level 2

In level 2, variables are introduced, using the new command **is**. This makes **echo** unnecessary, because we can repeat given input if we store it in a variable. So **echo** disappears and for **ask**, we need to store the answer the user gives in a variable ourselves using **... is ask ...**. This means that **ask** cannot be used separately anymore, it is always combined with **is**, to make sure the variable is stored. For example:

```
name is ask What is your name?
```

The command **sleep** is also introduced. Thus, we have the following grammar for any $c \in Commands$, with $s \in A^+$ for some alphabet A, $n \in \mathbb{Z}$ and a semicolon instead of a newline. Now that we have variables, we also have a set of possible variable names, or locations in which values can be stored, called Loc. Here, $v \in Loc$.

$$\text{c} ::= \mathbf{print}\, s \mid v\, \mathbf{is}\, \mathbf{ask}\, s \mid v\, \mathbf{is}\, s \mid \mathbf{turn}\, o \mid \mathbf{forward}\, o \mid \mathbf{sleep}\, o \mid c; c$$
$$o ::= \epsilon \mid n \mid v$$

The grammar shows that **turn** and **forward** can now be used with a variable, an integer or nothing, which is different from level 1. For **turn**, 'left' and 'right' are not used anymore.

Now that we have variables, it is time to take a better look at the type system. In level 2, we only work with strings and integers.

$$T ::= \text{String} \mid \text{Integer}$$

An element of a certain type is an element in a certain set.

$$sem\colon \text{Type} \to \text{Set}$$

In the case of a string, it is an element of some alphabet A^* and in the case of an integer, it is an element of $\mathbb{Z}$.

$$sem(\text{String}) = A^*$$
$$sem(\text{Integer}) = \mathbb{Z}$$

The memory Σ_2 is where we find the values of our variables. A given location or variable name $v \in Loc$ will map to an element of one of the aforementioned sets. The incomplete arrow $\rightharpoonup$ means that it is a partial map, explained in Section 3.3.1. The $\coprod$ sign means 'coproduct', or disjoint union, also explained further in Section 3.3.2. This definition of the memory Σ_2 makes sure that a variable will consist of a value of one type, string or integer in our case, but not both, so it will be an element in the alphabet A^* or in the set $\mathbb{Z}$, but not both.

$$\Sigma_2 = [Loc \rightharpoonup \coprod_{t \in Type} sem(t)]$$

We define that E_2 is an environment which includes E_1 except $echo\_possible$, and also includes some new functions. The functions we still have from level 1 are $print, ask, turn, forward$ and $draw$. The first new function is upd, which we need for the command **is**, the assignment of values to variables, or the 'update' function. It needs the variable name/location, the value and the memory and returns the new memory in which the given value then should be stored in the right location. It can be stored as a string, it can be converted to an integer later if needed.

$$upd\colon Loc \times A^+ \times \Sigma_2 \to \Sigma_2$$

The second new function is $sleep$, for command **sleep**. It takes an integer and should make sure the program 'sleeps' (does nothing) for the amount of time indicated by the integer.

$$sleep\colon \mathbb{Z} \times E_2 \to E_2$$

The third and fourth new functions are $substStr$ and $substInt$. When we want to print a string, the variables need to be replaced by what is stored in that variable. The function thus needs a string and the memory as input and gives back a new string, in which the variables should be substituted by their values. This means that $substStr$ takes a string in which there may be one or more variables used and memory and returns the new string in which those variables are replaced. When a variable is used with $\mathbf{turn}, \mathbf{forward}$ or $\mathbf{sleep}$, it should be replaced by an integer. This means that $substInt$ should get exactly one variable name/location and, together with the memory, returns an integer.

$$substStr\colon A^* \times \Sigma_2 \to A^*$$

$$substInt\colon Loc \times \Sigma_2 \to \mathbb{Z}$$

We have three commands which can optionally be used with a number, **turn, forward** and **sleep**. To make sure the functions we use for these commands will always get a number, we can still use rules (num), ϵ_{turn} and $\epsilon_{forward}$ from level 1, and add the following rule.

$$\frac{}{\epsilon \Downarrow 1}\ (\epsilon_{sleep})$$

The rules we already have for the different commands used in level 1 need to be adjusted, because we do not have the **echo** command anymore, which means we do not need rules of this form anymore:

$$\frac{}{< e, s, b, c > \Downarrow < e', s', b' >}$$

because the s was to remember the last input the user gave us. We now have to take variables into account. A state in level 2 is of the form:

$$S_2 = E_2 \times \mathbb{B} \times \Sigma_2$$

Now, for every command c, we want to have rules of the following form, with $e, e' \in E_2$, $b, b' \in \mathbb{B}$ to indicate whether there already is a turtle or not (1 means there is a turtle, 0 means there is not) and $\sigma, \sigma' \in \Sigma_2$.

$$\frac{}{< e, b, \sigma, c > \Downarrow < e', b', \sigma' >}$$

The adjusted rules from level 1 are as follows. For a command c of the form $\mathbf{print}\, x$, we cannot immediately print the string x with the function $print$, but we need to replace variables by their values first. For that, we will introduce a substitution rule $(substStr_{lvl2})$ later, for now we can say that the string x, together with the memory σ should evaluate to a certain string x' and that should be the one we print.

$$\frac{< x, \sigma > \Downarrow_{String} x' \qquad print(x', e) = e'}{< e, b, \sigma, \mathbf{print}\, x > \Downarrow < e', b, \sigma >} \; (print_{lvl2})$$

From this level on, **ask** can no longer be used on its own, but only always in combination with **is**. We can thus see **is ask** as one construction, instead of a combination of two constructions. Normally, we would first want to substitute the variables in the question for their values, but Hedy does not support this functionality yet, so in this description of the semantics, I will not include this, the description should correspond with the current behaviour of Hedy. Later on, an addition might be necessary, similar to the addition in the rule for **print**. Thus, if the command c is of the form $x\, \mathbf{is\, ask}\, y$, we skip the substitution step and immediately use the ask function, which outputs the given question y and reads an input from the user. This input should then be stored in the given variable location x, for this we use the update function upd. The state we end up in has a changed environment and changed memory.

$$\frac{ask(y, e) = (s, e') \qquad upd(x, s, \sigma) = \sigma'}{< e, b, \sigma, x\, \mathbf{is\, ask}\, y > \Downarrow < e', b, \sigma' >} \; (ask_{lvl2})$$

When the command c is of the form $\mathbf{turn}\, o$, and $b = 1$, meaning that this is not the first turtle command used, so there is a turtle already and we do not need to use $draw$ to initialise one, we first need to make sure the command is used with an integer, just like in level 1. The difference is that we can now also have a variable in which an integer is stored. This means that we need to see if o is empty, an integer or a variable with a value of type integer. For this, we will define a rule $(substInt_{lvl2})$ later, for now, we can say that o, together with the memory σ should evaluate to a certain integer n, similar to what we did for the new $(print_{lvl2})$ rule.

$$\frac{< o, \sigma > \Downarrow_{Integer} n \qquad turn(n, e) = e'}{< e, 1, \sigma, \mathbf{turn}\, o > \Downarrow < e', 1, \sigma >} \; (turn1_{lvl2})$$

When the command c is of the form $\mathbf{turn}\, o$ and $b = 0$, meaning that there have not been turtle commands earlier, so we do not have a turtle yet, we first need to use $draw$ to initialise the turtle. Then we can use the new $(turn_{lvl2})$ rule we just defined, similar to what we did in level 1.

$$\frac{draw(e) = e' \qquad < e', 1, \sigma, \mathbf{turn}\, o > \Downarrow < e'', 1, \sigma >}{< e, 0, \sigma, \mathbf{turn}\, o > \Downarrow < e'', 1, \sigma >} \; (turn0_{lvl2})$$

When the command c is of the form $\mathbf{forward}\, o$ and $b = 1$, we do not need $draw$ and just like for $\mathbf{turn}$, we first need to make sure the command is used with an integer. This can again be a variable, different from level 1. As already mentioned, later on, we will define a rule $(substInt_{lvl2})$ that substitutes the variable for its value of type integer. Here, we can say that o, together with the memory σ should evaluate to a certain integer n, which can then be used

for the $forward$ function in this case.

$$\frac{< o, \sigma > \Downarrow_{Integer} n \qquad forward(n, e) = e'}{< e, 1, \sigma, \textbf{forward}\, o > \Downarrow < e', 1, \sigma >} \; (forward1_{lvl2})$$

When the command c is of the form $\textbf{forward}\, o$ and $b = 0$, meaning that there have not been turtle commands earlier, so we do not have a turtle yet, we first need to use $draw$ to initialise the turtle. Then we can use the new $(forward_{lvl2})$ rule we just defined, similar to what we did for $\textbf{turn}$.

$$\frac{draw(e) = e' \qquad < e', 1, \sigma, \textbf{forward}\, o > \Downarrow < e'', 1, \sigma >}{< e, 0, \sigma, \textbf{forward}\, o > \Downarrow < e'', 1, \sigma >} \; (forward0_{lvl2})$$

For sequential composition, the only change is that the states are different, instead of the input string we now have memory, but everything still works the same as in level 1.

$$\frac{< e, b, \sigma, c > \Downarrow < e', b', \sigma' > \qquad < e', b', \sigma', c' > \Downarrow < e'', b'', \sigma'' >}{< e, b, \sigma, c; c' > \Downarrow < e'', b'', \sigma'' >} \; (composition_{lvl2})$$

The new rules for this level are as follows. When the command c is of the form $x\, \textbf{is}\, y$, the memory σ needs to be changed such that at location x, we have the value y. For this, we use the upd function. Only the memory changes.

$$\frac{upd(x, y, \sigma) = \sigma'}{< e, b, \sigma, x\, \textbf{is}\, y > \Downarrow < e, b, \sigma' >} \; (is_{lvl2})$$

When the command c is of the form $\textbf{sleep}\, o$, similar to $\textbf{turn}$ and $\textbf{forward}$, we first need to make sure the o is an integer. It can be empty, in which case it will evaluate to 1, given by the (ϵ_{sleep}) rule, or it is an integer, in which case we can use (num), but if it is a variable, we need to use $(substInt_{lvl2})$, which we will define soon. For this rule, similar to the rules for $\textbf{turn}$ and $\textbf{forward}$, we can just state that the o, together with the memory σ evaluates to a certain integer n. This integer can than be used for the function $sleep$.

$$\frac{< o, \sigma > \Downarrow_{Integer} n \qquad sleep(n, e) = e'}{< e, b, \sigma, \textbf{sleep}\, o > \Downarrow < e', b, \sigma >} \; (sleep_{lvl2})$$

The final rules we need for this level, already mentioned before and long waited for, are $substStr_{lvl2}$ and $substInt_{lvl2}$. $substStr_{lvl2}$ makes sure that all the variables that may be present in string x will be replaced by their values and gives back the complete string. $substInt_{lvl2}$ however, takes one variable name/location x and gives the integer value.

$$\frac{substStr(x, \sigma) = x'}{< x, \sigma > \Downarrow_{String} x'} \; (substStr_{lvl2}) \qquad \frac{substInt(x, \sigma) = x'}{< x, \sigma > \Downarrow_{Integer} x'} \; (substInt_{lvl2})$$

Level 3

In level 3, lists are introduced. Lists can be used with the new commands **at random**, which chooses a random element from a list, **add to** which adds an element to a list and **remove from** which removes an element from a list. The addition of lists also changes how commands we already had can be used. Since the command **at random** chooses a random element from a list, this can be done in combination with **print** for example:

```
animals is dog, cat, kangaroo
print animals at random
```

Running this Hedy program, only one of the three animals in the list will be printed. We can not yet print a whole list, but we can print a random element of a list like that. We would want **ask** to work in a similar way, but it does not work yet in the current implementation of Hedy, so we decided to make this description of the semantics of Hedy correspond with how Hedy works now and thus the **is ask** construction will only work with a string and not with **at random**. The commands **turn, forward** and **sleep** can also be used in combination with lists and **at random**, if the list consists of integers. What type of elements a lists should consist of is not specified in the grammar, this will be made sure later, in the type system. It is also always needed to store the list in a variable location, before using it with $\mathbf{at\,random}$, $\mathbf{add\,to}$ or $\mathbf{remove\,from}$. For example, this is not possible:

```
dog, cat, kangaroo at random
```

This means that we say in the grammar that the commands $\mathbf{at\,random}$, $\mathbf{add\,to}$ and $\mathbf{remove\,from}$ work with variables and later on in the type system, we make sure that the values of the variables are lists. This gives us the following grammar for level 3.

$$
\begin{aligned}
c &::= \mathbf{print}\, u \mid v\, \mathbf{is\,ask}\, s \mid v\, \mathbf{is}\, u \mid \mathbf{turn}\, o \mid \mathbf{forward}\, o \mid \mathbf{sleep}\, o \\
&\mid \mathbf{add}\, s\, \mathbf{to}\, v \mid \mathbf{remove}\, s\, \mathbf{from}\, v \mid c; c \\
o &::= \epsilon \mid n \mid v \mid v\, \mathbf{at\,random} \\
u &::= s \mid n \mid v\, \mathbf{at\,random} \\
l &::= m, m \mid l, m \\
m &::= n \mid s
\end{aligned}
$$

Here, as before, s is a string, so $s \in A^+$ for some alphabet A, $n \in \mathbb{Z}$, $v \in Loc$ and now we also have $l \in \mathrm{List}\, T$ (lists with elements of type T, in this level list of strings or lists of integers). We have to add lists to the type system of level 2 for the type system of level 3. We now have strings, integers and lists.

$$T ::= \mathrm{String} \mid \mathrm{Integer} \mid \mathrm{List}\, T$$

An element of a certain type is in a certain set.

$$sem\colon \mathrm{Type} \to \mathrm{Set}$$

In the case of a string, it is an element of A^+, for some alphabet A. In the case of an integer, it is an element of $\mathbb{Z}$. A list is of type $\mathbb{Z}^*$ if the elements of the list are integers and $(A^+)^*$ if

the elements of the list are strings. For the lists, this can be generalised to $sem(T)^*$ in which T is an integer or a string.

$$\begin{aligned} sem(\text{String}) &= A^* \\ sem(\text{Integer}) &= \mathbb{Z} \\ sem(\text{List}\, T) &= sem(T)^* \end{aligned}$$

The memory Σ_3 is where we find the values of our variables. The notation of the definition of Σ_3 is exactly the same as for level 2, Σ_2, but since we have added lists to the set $Type$, the definition is not exactly the same. A given location or variable name $v \in Loc$ will map to an element of one of the aforementioned sets, so it is an element of $A^*, \mathbb{Z}$ or $sem(T)^*$, because the value of the variable then is of type string, integer or list respectively. As mentioned in level 2, $\rightharpoonup$ denotes a partial map and $\coprod$ means coproduct or disjoint union, explained in Section 3.3.1 and Section 3.3.2 respectively. This definition of the memory Σ_3 makes sure that a variable will consist of a value of one type, string, integer or list in this case, but not multiple types. It also makes sure that the lists consist of only integers or only strings, so the lists cannot consists of elements of different types. In the grammar, we did not specify this yet, but this is now solved in the type system.

$$\Sigma_3 = [Loc \rightharpoonup \coprod_{t \in Type} sem(t)]$$

The type system also includes the rules given below. The first rule states the following. If a random element of a list v, given by $v\,\textbf{at random}$, is of type T, then the list v is of type $\text{List}\, T$. The second rule states that **print** is used with a string. The third rule says that using **is ask**, both the variable name and the question are of type string. This is correct, because $Loc \subset A^*$. The fourth rule says that using **is**, both the variable name and the value are of type string. This is correct, because the variable name is an element of Loc again, and $Loc \subset A^*$. The value can be stored as a string and later on be converted to an integer or list if needed. The fifth, sixth and seventh rule make sure that respectively **turn**, **forward** and **sleep** can only be used with an integer. The eighth and ninth rule make sure that **add to** and **remove from** can only be used with a list and a string to add to or remove from that list. The elements of lists can be converted to integers when needed (and if we can convert an element of a list to an integer, we can do that with the other elements of the same list as well). The tenth rule is a very general rule, with the eleventh rule as an example using a list instead of the general T. For more information on type systems, see Section 3.4.

$$\frac{v : \text{List}\, T \in \Gamma}{\Gamma \vdash v\,\textbf{at random} : T} \qquad \frac{u : \text{String} \in \Gamma}{\Gamma \vdash \textbf{print}\, u} \qquad \frac{s, t : \text{String} \in \Gamma}{\Gamma \vdash s\,\textbf{is ask}\, t} \qquad \frac{s, u : \text{String} \in \Gamma}{\Gamma \vdash s\,\textbf{is}\, u}$$

$$\frac{u : \text{Integer} \in \Gamma}{\Gamma \vdash \textbf{turn}\, u} \qquad \frac{u : \text{Integer} \in \Gamma}{\Gamma \vdash \textbf{forward}\, u} \qquad \frac{u : \text{Integer} \in \Gamma}{\Gamma \vdash \textbf{sleep}\, u} \qquad \frac{s : \text{String},\, l : \text{List}\, T \in \Gamma}{\Gamma \vdash \textbf{add}\, s\, \textbf{to}\, l}$$

$$\frac{s : \text{String},\, l : \text{List}\, T \in \Gamma}{\Gamma \vdash \textbf{remove}\, s\, \textbf{from}\, l} \qquad \frac{v : T \in \Gamma}{\Gamma \vdash v : T} \qquad \frac{l : \text{List}\, T \in \Gamma}{\Gamma \vdash l : \text{List}\, T}$$

We define E_3 as an environment which includes the same functions as E_2 and also includes some new functions. The functions we still have from level 1 and 2 are $print$, ask, $turn$, $forward$, $draw$, upd, $sleep$, $substStr$ and $substInt$. The update function has to be adjusted though, since we now have lists. For level 2, we said:

$$upd\colon Loc \times A^+ \times \Sigma_2 \to \Sigma_2$$

Thus, every variable was assigned as a string and we could convert it to an integer if needed later. We cannot do this with lists, we are now actually going to assign values as the corresponding type (integer, string or list) to the variable names/locations. We get:

$$upd\colon Loc \times sem(T) \times \Sigma_3 \to \Sigma_3$$

The following new functions are needed for the commands that have to do with lists. The function $random$ takes a list and the environment and returns one of its elements and the new environment. The function $addto$ takes an element and a variable location (where a list should be stored) and the memory, and maps to a new list (not the location!) in which this element is added. Finally $remove$ takes an element, a variable location (where a list should be stored) and the memory as well and maps to a list in which this element is removed if it was present in the initial list.

$$\begin{aligned} random\colon sem(T)^* \times E_3 &\to sem(T) \times E_3 \\ addto\colon A^+ \times Loc \times \Sigma_3 &\to (A^+)^* \\ remove\colon A^+ \times Loc \times \Sigma_3 &\to (A^+)^* \end{aligned}$$

A state in level 3 is of the form $S_3 = E_3 \times \mathbb{B} \times \Sigma_3$
For every command c, we want to have rules of the form: $\dfrac{}{< e, b, \sigma, c > \Downarrow < e', b', \sigma' >}$

Here, $e, e' \in E_3$, $b, b' \in \mathbb{B}$ to indicate whether there already is a turtle or not. 1 means there is a turtle, 0 means there is not, and $\sigma, \sigma' \in \Sigma_3$.
The new rules are as follows. For a command c of the form $\mathbf{add}\, s\, \mathbf{to}\, v$, if the function $addto$ given the element to add s, the location of the list v and the memory σ as input returns a list l', and function upd given input l, l' and the memory σ returns memory σ', then this σ' is the new memory in our state. The other elements of the state are unchanged.

$$\frac{addto(s, v, \sigma) = l' \qquad upd(v, l', \sigma) = \sigma'}{< e, b, \sigma, \mathbf{add}\, s\, \mathbf{to}\, v > \Downarrow < e, b, \sigma' >} \; (addto_{lvl3})$$

For a command c of the form $\mathbf{remove}\, s\, \mathbf{from}\, v$, this works similar to the formal one. The function $remove$ takes s, v and σ and returns a new list and with the function upd this new list is stored in the location of the old list and returns the changed memory. This changed memory σ' is the only thing that changes in the state.

$$\frac{remove(s, v, \sigma) = l' \qquad upd(v, l', \sigma) = \sigma'}{< e, b, \sigma, \mathbf{remove}\, s\, \mathbf{from}\, v > \Downarrow < e, b, \sigma' >} \; (remove_{lvl3})$$

For any command used with $\mathbf{at\,random}$, we need a rule that can evaluate $v\,\mathbf{at\,random}$ to a string or integer, depending on the command it is used in combination with. The rules for the other commands do not need to change for this, because they use the fact that certain things need to evaluate to something else of a certain type in order to evaluate the commands. For example, if it is $\mathbf{turn}$, $\mathbf{forward}$ or $\mathbf{sleep}$, it has to evaluate to an integer (if not, it will result in an HedyException) and this is also given in the rules (see level 2). The rule below denotes that $v\,\mathbf{at\,random}$ evaluates to t of type T if the function $random$ given v returns t. The environment e is also needed, because the random generator depends on the environment.

$$\frac{random(v, e) = (t, e')}{v\,\mathbf{at\,random} \Downarrow_T t}\ (random_{lvl3})$$

Level 4

In level 4, there are new syntax rules (we need to use ‘’ around text when we print or ask something, except for variables). For example, in level 3, this would be a correct program:

```
print You do not need to use quotation marks yet
answer is ask What do we not yet need to use?
print We do not need to use answer
```

In the final line, the word `answer` will be replaced by the input given by the user. In level 4, to achieve a similar program, this would be the correct syntax:

```
print ‘You need to use quotation marks from now on!’
answer is ask ‘What do we need to use from now on?’
print ‘We need to use ’ answer
```

If the quotes would not be used, it would result in a HedyException. If the closing quote in the final line would be after `answer` instead of before `answer`, it would not be replaced, but printed literally. Everything between the quotes is seen as plain text and outside the quotes, the words should be names of variables. This way, words used as variable names can also be printed. For example, in level 3,

```
name is Hedy
print my name is name
```

would give the following output:

```
my Hedy is Hedy
```

but in level 4, because of the quotation marks, we can use the following code:

```
name is Hedy
print ‘my name is ’ name
```

which will result in:

```
my name is Hedy
```

The only change made in level 4 are thus the quotation marks, to make a difference between plain text and variables when printing or asking something, which is a change in the syntax. This means there are not changes in the semantics. $E_4 = E_3$, $\Sigma_4 = \Sigma_3$, we have the same rules and axioms. It does mean though, that the function $substStr$ will work slightly different. It still takes a string and the memory and returns a new string in which variables should be replaced by their values, but the function does not have to check the whole string for variables anymore. The parts between quotes can be skipped and the other parts should contain variables and be replaced by their values. This does not change anything for the derivation rules we have, but it does change the implementation of $substStr$, which is not specified here, but it can be found in the in the appendix in the implementation in Dafny.

Level 5

In level 5, **if** and **if else** is added to the possible commands. Command **is** can now not only be used for assignment, but also for comparison of strings. It is also possible to check if something is an element of a list with **in**. For example:

```
name is ask 'what is your name?'
if name is Hedy print 'cool!' else print 'meh'
```

Here, **is** is used as before, for assignment, in the first line and in the second line, **is** is used for comparison. In the next example, we have **is** again for assignment and the if-condition is now using **in** to see if something is an element of a list:

```
pretty_colors is green, yellow
color is ask 'What is your favorite color?'
if color in pretty_colors print 'pretty!'
else print 'meh'
```

The addition of these possibilities gives us the following new grammar for level 5:

$$
\begin{aligned}
c &::= \mathbf{print}\, u \mid v\, \mathbf{is\, ask}\, s \mid v\, \mathbf{is}\, u \mid \mathbf{turn}\, o \mid \mathbf{forward}\, o \mid \mathbf{sleep}\, o \mid \mathbf{add}\, s\, \mathbf{to}\, v \mid \mathbf{remove}\, s\, \mathbf{from}\, v \\
&\quad \mid \mathbf{if}\, b\, \mathbf{then}\, c \mid \mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, c \mid c; c \\
o &::= \epsilon \mid v \mid n \mid v\, \mathbf{at\, random} \\
u &::= s \mid n \mid v\, \mathbf{at\, random} \\
m &::= n \mid s \\
l &::= m, m \mid l, m \\
b &::= s\, \mathbf{in}\, v \mid m\, \mathbf{is}\, m
\end{aligned}
$$

Here, as before, s is a string, so $s \in A^+$ for some alphabet A, $n \in \mathbb{Z}$, $v \in Loc$, $l \in \mathrm{List}\, T$ (lists with elements of type T) and now we also have $b \in \mathbb{B}$. In the syntax of Hedy, **then** is actually not used for **if then else**, but only **if else**. To make it completely clear what the if-condition is and what the command to be executed is (if the condition is true), I have added a **then**. Obviously, we have a new type now, the boolean, which we need for the if-conditions. This gives us:

$$T ::= \mathrm{String} \mid \mathrm{Integer} \mid \mathrm{List}\, T \mid \mathrm{Boolean}$$

Just like with the other types, an element of type boolean is the element of a set, in this case:

$$sem(\mathrm{Boolean}) = \{\mathrm{tt}, \mathrm{ff}\}$$

To denote that the if-conditions always have to be of type boolean, we have the following rule, in addition to the rules given in level 3.

$$\frac{b : \mathrm{Boolean} \in \Gamma}{\Gamma \vdash \mathbf{if}\, b}$$

We define E_5 as an environment which includes all functions from $E_4 = E_3$, and also includes the following new functions. The function eq takes two elements of type T, so of the set $sem(T)$ and returns a boolean, true if they are equal and false if they are not. A HedyException occurs if the types of the elements make them incomparable. The function $elem$ takes an element and a list and returns true if the element occurs in the list and false if it does not.

$$eq\colon sem(T) \times sem(T) \to \mathbb{B}$$
$$elem\colon sem(T) \times sem(T)^* \to \mathbb{B}$$

A state in level 5 is of the form:

$$S_5 = E_5 \times \mathbb{B} \times \Sigma_5$$

We can use the rules from level 3 (= 4) and we add some rules, of the same form as before, for our new commands. The new rules are as follows. For the evaluation of the if-conditions, we need rules for both forms that the if-conditions can have at this level. The two possible forms the if-condition can have are $x\,\mathbf{is}\,y$ for equality comparison and $x\,\mathbf{in}\,y$ for checking the occurrence of an element in a list, both should evaluate to true or false, depending on the output of the corresponding function, eq and $elem$ respectively. The two rules are:

$$\frac{eq(x,y) = b}{x\,\mathbf{is}\,y \Downarrow b}\ (eq_{lvl5}) \qquad \frac{elem(x,y) = b}{x\,\mathbf{in}\,y \Downarrow b}\ (elem_{lvl5})$$

Now that we can evaluate the if-condition, we still need rules for what to do if we have an if-statement and we know the outcome of the if-condition. In the case that the if-condition evaluates to false, and the if-statement is of the form $\mathbf{if}\,b\,\mathbf{then}\,c$, without an $\mathbf{else}$, we should not execute command c, which means that we do nothing and our state does not change. This gives us the following rule.

$$\frac{b \Downarrow \mathrm{ff}}{< e, b_t, \sigma, \mathbf{if}\,b\,\mathbf{then}\,c > \Downarrow < e, b_t, \sigma >}\ (if - ff_{lvl5})$$

In the case that the if-condition evaluates to true however, and the if-statement is of the form $\mathbf{if}\,b\,\mathbf{then}\,c$ (without an $\mathbf{else}$), we have to execute c and the state will change depending on c. This gives us the following rule.

$$\frac{b \Downarrow \mathrm{tt} \qquad < e, b_t, \sigma, c > \Downarrow < e', b_t', \sigma' >}{< e, b_t, \sigma, \mathbf{if}\,b\,\mathbf{then}\,c > \Downarrow < e', b_t', \sigma' >}\ (if - tt_{lvl5})$$

In the case that the if-condition evaluates to false, but now there is an $\mathbf{else}$, so the form of the if-statement is $\mathbf{if}\,b\,\mathbf{then}\,c\,\mathbf{else}\,c'$, we have to execute c'. The state will change depending on c'. This gives us the following rule.

$$\frac{b \Downarrow \mathrm{ff} \qquad < e, b_t, \sigma, c' > \Downarrow < e', b_t', \sigma' >}{< e, b_t, \sigma, \mathbf{if}\,b\,\mathbf{then}\,c\,\mathbf{else}\,c' > \Downarrow < e', b_t', \sigma' >}\ (if - else - ff_{lvl5})$$

The final case is that the if-condition evaluates to true and the if-statement is of the form $\mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, c'$. We need to execute the c and not the c'. The state will change depending on c. This gives us the following rule.

$$\frac{b \Downarrow \mathrm{tt} \qquad < e, b_t, \sigma, c > \Downarrow < e', b_t', \sigma' >}{< e, b_t, \sigma, \mathbf{if}\, b\, \mathbf{then}\, c\, \mathbf{else}\, c' > \Downarrow < e', b_t', \sigma' >}\ (if - else - tt_{lvl5})$$

To illustrate what a derivation tree / prooftree could look like using these rules, we have the following example. This Hedy program:

```
name is Hedy
if name is Hedy print 'nice'
```

assuming that the user would give the input-string 'Hedy', gives this tree:

$$\frac{\dfrac{upd(\mathsf{name}, \mathsf{Hedy}, \sigma) = \sigma'}{< e, 0, \sigma, \mathsf{name}\, \mathbf{is}\, \mathsf{Hedy} > \Downarrow < e, 0, \sigma' >}\ (is_{lvl2}) \qquad ***}{< e, 0, \sigma, \mathsf{name}\, \mathbf{is}\, \mathsf{Hedy}; \mathbf{if}\, \mathsf{name}\, \mathbf{is}\, \mathsf{Hedy}\, \mathbf{then\ print}\ \text{'nice'} > \Downarrow < e', 0, \sigma' >}\ (composition_{lvl2})$$

The complete derivation tree does not fit on the page. Replace $***$ by the following subtree:

$$\frac{\dfrac{eq(\mathsf{name}, \mathsf{Hedy}) = \mathrm{tt}}{\mathsf{name}\, \mathbf{is}\, \mathsf{Hedy} \Downarrow \mathrm{tt}}\ (eq_{lvl5}) \qquad \dfrac{< \text{'nice'}, \sigma' > \Downarrow_{\mathrm{String}} \text{'nice'} \qquad print(\text{'nice'}, e) = e'}{< e, 0, \sigma', \mathbf{print}\ \text{'nice'} > \Downarrow < e', 0, \sigma' >}\ (print_{lvl2})}{< e, 0, \sigma', \mathbf{if}\, \mathsf{name}\, \mathbf{is}\, \mathsf{Hedy}\, \mathbf{then\ print}\ \text{'nice'} > \Downarrow < e', 0, \sigma' >}\ (if - tt_{lvl5})$$

Note that the left subtree evaluates a command including **is** with the (is_{lvl2}) rule, for assignment, while in the right subtree, we need the (eq_{lvl5}) rule for equality comparison, for a command including **is**. Also note that the resulting state after evaluating the first line of the code in the left subtree is the same as the initial state in the right subtree evaluating the second line of the code.

Now, we can evaluate all correct programs in Hedy code level 1-5 using derivation trees, constructed by our rules. If the evaluation of a Hedy program fails, there is an error in the code and an HedyException should occur.

5.3 Implementation of Semantics of Hedy in Dafny

Using the formal description of the semantics of level 1 to 5, the semantics could be implemented in Dafny. The goal is to generate tests. The Dafny code can be compiled to other languages, in this case, C# has been used. The Dafny code can be found in the appendix and both the Dafny and the C# code can be found on GitHub [Bol]. For the implementation in Dafny, *traits* have been used [ALN15]. Traits are abstract superclasses, which make it possible to describe behaviour common to several classes and to write code that abstracts over the particular classes involved. There is a separate file per level. The only difference between level 3 and 4 is the method substStr, which works slightly different because of the addition of quotes around text in level 4. Before we can use the Dafny code as reference implementation, we need to prove certain properties, so that we know for sure the Dafny code is correct. First attempts to proving those properties for level 1 can also be found on GitHub [Bol]. The properties we want to prove are the ones we want to test to make sure that the implementation of Hedy behaves in the way we want. If we prove those properties in the implementation in Dafny, we know these also hold in the compiled C# code and then we can use it as a reference implementation and generate test cases. The testcases should run in the generated C# code and be translated to Python to test the Hedy code. Then, the behaviours of the two implementations can be compared. Because of the proven properties, we know that the behaviour we get from running the testcases in C# is correct. If it does not correspond with the behaviour from the implementation of Hedy in Python, we know that there is a mistake in the implementation of Hedy, and preferably, even a counterexample can be given [CFRR22], to show the case in which the behaviour differs from the wanted behaviour.

6 Uncovered issues in Hedy

Creating a description of the semantics of Hedy already uncovered multiple issues.

1. In level 14, comparisons are added. Listing all the possible commands and their types, uncovered that the equality comparison and the inequality comparison could not be used for exactly the same types. The types allowed for equality (==) were only integer, float and string, while for not-equal (!=), also lists were possible.

2. When using an if/else-statement to create a variable which is a string in one case and a list in the other, making sure it will be the string-case, but afterward printing it with **at random** as a list, we get weird behavior. It seems that Hedy sees the string as a list of characters and it prints a random character of this string (including the quotes) instead of giving an error that it is not possible to print a string at random, that this can only be used for a list. At first, this issue seemed too hard (and not important enough) to fix, but it has been solved now.

3. While this was no problem for the command **forward**, the command **turn** could not be used with a negative number. This has been solved.

4. In level 3, lists can be used, with, among others, command **at random** to choose a random element of a list. This command would be used in combination with other commands. It should work in combination with **ask** for example, but this did not work, while it did in combination with **print** for example.

5. In level 10, **repeat** and **for** both worked, but from level 11 on, only **for** worked. After discussing this with the creator of Hedy, Felienne Hermans [Hera], the following decisions were made. The description of the semantics should correspond / be consistent with how Hedy works currently, with **repeat** not possible in level 11 and further. Later on, **repeat** should be added in level 11 and further, but that might take a while.

6. Similar to the fourth issue, in level 3, **at random** should, but could not, be used in combination with **sleep**. This issue has been solved already.

For a more detailed explanation of the issues (and their solutions), the issues can be found on GitHub [Herb]. Some of the issues have been solved already. The issues found while describing the semantics are issues #1813, #1924, #1927, #1938, #1942 and #2431 respectively.

For another small improvement on Hedy, creating a new issue was not necessary, but a new pull request could immediately be made to make the change: Listing the possible commands per level made clear that in level 14, not every new command was explained in the text. The comparisons $<$, $>$, $==$ and $!=$ had been explained, but $<=$ and $>=$ not yet, while these comparisons could also be used in this level. An explanation for these final two comparisons was added using a pull request, without first creating an issue. (Pull requests #1816 and #1820)

7 Conclusions and discussion

The goal of this research project was to help improve the Hedy programming language. Hedy is being developed for education, using a new approach inspired by natural language education. Hedy consists of multiple levels, starts very simple and adds programming concepts or syntax rules gradually. These multiple levels can all be seen as their own language and have their own semantics. The semantics of a language describes the behaviour of programs in that language or the language as a whole, it gives meaning to the syntax. Formally describing the semantics of a language helps to improve it, by finding unwanted or illogical behaviours. Dafny can be used to implement semantics and compile it to other programming languages, for a reference implementation. Having a reference implementation of how a language should work is helpful for the people still developing Hedy and might uncover issues. The research question that follows is: How can we uncover issues in the implementation of the Hedy programming language, by providing a formal description of its semantics and a formalised implementation?

As discussed in Section 6, issues have been uncovered, by describing the semantics of Hedy. First, describing the semantics informally and only checking which commands with which types are supported by every level, already helped to find issues. Then, describing the semantics formally, more issues were found. Hopefully the implementation of the semantics of Hedy in Dafny will also help find more issues in the future.

7.1 Future work

The Hedy language currently consists of 18 levels, of which now only the semantics of the first 5 levels have been described. To complete the description, the other levels can be described. It is also important to note that Hedy is still being improved, so that the transitions to new levels will be as easy as possible for children using it. This means that the description of the semantics given in this thesis will very likely not be accurate in the future. When expanding the description of the semantics of Hedy, not only the higher levels should be added, but possibly the first 5 levels should also be adjusted.

If there is a complete description on paper, a complete implementation in Dafny could be made. The goal would be to use this as a reference implementation, in which we can prove certain properties. When these properties are proven in the Dafny code, testcases can be generated, to run in the generated C# code. For this, another language could also be used, it does not necessarily have to be C#, Dafny can also generate Java for example. There would also still have to be made a translation to Python, because Hedy is implemented (and tested) with Python code. (Currently, Dafny cannot generate Python.) Then, ouputs/behaviours can be compared. Because of the proven properties, we know that the behaviour we get from running the testcases in C# is correct. If this behaviour does not correspond, we know there is a mistake in the implementation of Hedy. Preferably, a counterexample can be given [CFRR22], to show in which case the behaviour is different from the wanted behaviour.

7.2 Limitations

The approach chosen to answer the research question might not have been the easiest one. A reference implementation is helpful to test software, because of the certainty that this reference implementation is correct. To be sure this reference implementation is correct, certain properties need to be proven first. Proving these properties to make sure the reference implementation is correct, can be done in Dafny, but working with input and output makes this a non-trivial job. Another approach could have been to implement the semantics of Hedy in any language and compare this, without being sure that the implementation based on the semantics is error-free. When comparing the behaviours, there is a difference, it would not be clear in which implementation there is a problem. It would be necessary to check both implementations, to be sure which is the correct/wanted one. This might be a disadvantage, but not a very big one, because it seems this second approach might still be less work, since it skips the step of proving properties in the implementation based on the semantics.

A different limitation to this project is that the formal description of the semantics of Hedy and the implementation based on that in Dafny does not always correspond with the wanted/intended behaviour of Hedy. Instead, we chose to make sure the description and implementation correspond with the current behaviour of Hedy, but some details should change in the future. For example, the fourth issue in Section 6 made clear that the command **ask** can only be used to ask questions in plain text, but not in combination with **at random** or variables in general, while it is possible with the command **print**. The preferred behaviour of **ask** would be for it to work similar to **print**. We found this issue because of this project, when this issue is solved, the semantics description and implementation should be adapted. For the informal description of the semantics, even more things should probably be changed in the future. For example, in level 10, the **repeat** command and the **for** command both work, but from level 11 on, **repeat** does not work anymore, we have to use **for**. This is consistent with the informal description now, but later on, the developers of Hedy might want to change this and keep **repeat** possible. This would be something to take into account if this project would be continued and the higher levels would be described and implemented.

7.3 Conclusion

This thesis provides an approach to describing the formal big-step operational semantics of the Hedy programming language, a gradual programming language for education. Since Hedy consists of multiple levels, the semantics of one level can be described separately, as if it is an independent language. This way, the first 5 levels have been described. This helped uncover issues in the implementation of Hedy, listed in Section 6. Based on this description of the semantics of Hedy, the semantics of level 1 to 5 have also been implemented in Dafny. Hopefully, this will help improve Hedy further in the future.

References

[ACN+09] Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a syntax. In *Proceedings of Workshop on Foundations of Object-Oriented Languages*. Citeseer, 2009.

[ALN15] Reza Ahmadi, K Rustan M Leino, and Jyrki Nummenmaa. Automatic verification of dafny programs with traits. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*, pages 1–5, 2015.

[B+94] PETER Brusilovsky et al. Teaching programming to novices: A review of approaches and tools. 1994.

[BBD+12] Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen, and Mattias Ulbrich. The cost ic0701 verification competition 2011. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software*, pages 3–21, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BM17] Roberto Bruni and Ugo Montanari. *Models of computation*. Springer, 2017.

[Bol] Julia Bolt. Hedy in dafny. https://github.com/Felienne/hedy/semantics/Dafny. [Online; accessed 23-June-2022].

[CFRR22] Aleksandar Chakarov, Aleksandr Fedchin, Zvonimir Rakamarić, and Neha Rungta. Better counterexamples for dafny. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 404–411, Cham, 2022. Springer International Publishing.

[CO16] Walter Cazzola and Diego Mathias Olivares. Gradually learning programming supported by a growable programming language. *IEEE Transactions on Emerging Topics in Computing*, 4(3):404–415, 2016.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[EP10] Richard J. Enbody and William F. Punch. Performance of python cs1 students in mid-level non-python cs courses. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, page 520–523, New York, NY, USA, 2010. Association for Computing Machinery.

[EPM09] Richard J. Enbody, William F. Punch, and Mark McCullen. Python cs1 as preparation for c++ cs2. *SIGCSE Bull.*, 41(1):116–120, mar 2009.

[FFFK04] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The teachscheme! project: Computing and programming for every student. *Computer Science Education*, 14(1):55–77, 2004.

[Fre] Matt Frederikson. Induction and semantics in dafny. https://www.cs.cmu.edu/~mfredrik/15414/lectures/11-notes.pdf accessed at 01/06/2022.

[Geu08] Herman Geuvers. Introduction to type theory. In *International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development*, pages 1–56. Springer, 2008.

[HA17] Felienne Hermans and Marlies Aldewereld. Programming is writing is programming. In *Companion to the First International Conference on the Art, Science and Engineering of Programming*, Programming '17, New York, NY, USA, 2017. Association for Computing Machinery.

[Hera] Felienne Hermans. https://felienne.com. [Online; accessed 22-June-2022].

[Herb] Felienne Hermans. Hedy. https://github.com/Felienne/hedy. [Online; accessed 23-May-2022].

[Herc] Felienne Hermans. Hedy - a gradual programming language. https://hedycode.com. [Online; accessed 23-May-2022].

[Her20] Felienne Hermans. Hedy: A gradual language for programming education. In *Proceedings of the 2020 ACM conference on international computing education research*, pages 259–270, 2020.

[KMS+11] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st verified software competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods*, pages 154–168, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[KP05] Caitlin Kelleher and Randy Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37(2):83–137, jun 2005.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[lia] https://liacs.leidenuniv.nl. [Online; accessed 30-May-2022].

[LKT09] John C. Lusth, Nicholas A. Kraft, and James Tacey. Language subsetting via reflection and overloading. In *2009 39th IEEE Frontiers in Education Conference*, pages 1–6, 2009.

[LM10] K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, pages 112–126, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Mica] Microsoft. Dafny. https://github.com/dafny-lang/dafny. [Online; accessed 23-May-2022].

[Micb] Microsoft. Dafny documentation. https://dafny.org/dafny/. [Online; accessed 23-May-2022].

[MPS06] Linda Mannila, Mia Peltomäki, and Tapio Salakoski. What about a simple language? analyzing the difficulties in learning to program. *Computer Science Education*, 16(3):211–227, 2006.

[PDP19] Jean-Philippe Pellet, Amaury Dame, and Gabriel Parriaux. How beginner-friendly is a programming language? a short analysis based on java and python examples. 11 2019.

[Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[Pie04] Benjamin C Pierce. *Advanced topics in types and programming languages*. MIT press, 2004.

[Por18] Scott Portnoff. The introductory computer programming course is first and foremost a language course. *ACM Inroads*, 9(2):34–52, 2018.

[Shn77] Ben Shneiderman. Teaching programming: A spiral approach to syntax and semantics. *Computers & Education*, 1(4):193–197, 1977.

[SJP09] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 148–172, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[SKB+13] Pratim Sengupta, John S Kinnebrew, Satabdi Basu, Gautam Biswas, and Douglas Clark. Integrating computational thinking with k-12 science education using agent-based computation: A theoretical framework. *Education and information technologies*, 18(2):351–380, 2013.

[Tho91] Simon Thompson. *Type theory and functional programming*. Addison Wesley, 1991.

[VJV13] Carlos Vega, Camilo Jiménez, and Jorge Villalobos. A scalable and incremental project-based learning approach for cs1/cs2 courses. *Education and Information Technologies*, 18(2):309–329, 2013.

[WBH+16] David Weintrop, Elham Beheshti, Michael Horn, Kai Orton, Kemi Jona, Laura Trouille, and Uri Wilensky. Defining computational thinking for mathematics and science classrooms. *Journal of science education and technology*, 25(1):127–147, 2016.

[Wik22] Wikipedia contributors. Disjoint union — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Disjoint_union&oldid=1069045918, 2022. [Online; accessed 13-June-2022].

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[WX18] Jacques Wainer and Eduardo C. Xavier. A controlled experiment on python vs c for an introductory programming course: Students' outcomes. *ACM Trans. Comput. Educ.*, 18(3), aug 2018.

[Z3g] https://github.com/Z3Prover/z3 accessed at 24/05/2022.

Appendix

A: Dafny implementation of Level 1

```
// Level 1

datatype Com1 = Print(string)
              | Ask(string)
              | Echo(string)
              | Turn(D)
              | Forward(P)
              | Seq(Com1, Com1)

// Used for command turn
datatype D = Empty
           | Left
           | Right

// Used for command forward
datatype P = Empty
           | N(int)

datatype Maybe<T> = Nothing | Just(elem: T)

trait Env1
{
    var store: Maybe<string> // remember the answer after ask
    var turtle_exists: bool
    method Print(output: string) modifies this
    method Ask(question: string) returns (input: string) modifies
       this
    method Error(exception: string) modifies this // echo without ask
    method Turn(angle: int) modifies this
    method Forward(step: int) modifies this
    method Draw() modifies this
}

type Config1 = (Com1, Env1)
type Derivation1 = seq<Config1>

// Used for command turn
function method bigstep_D_level1(d: D): int
{
    match(d){
        case Empty => -90
        case Left => 90
        case Right => -90
    }
}

```

```
46 // Used for command forward
47 function method bigstep_P_level1(p: P): int
48 {
49     match(p){
50         case Empty => 50
51         case N(n) => n
52     }
53 }
54
55 method bigstep_level1(c: Com1, e: Env1)
56     modifies e
57 {
58     match(c){
59         case Print(x) =>
60             e.Print(x);
61         case Ask(x) =>
62               var answer := e.Ask(x);
63             e.store := Just(answer);
64         case Echo(x) =>
65             match(e.store) {
66                 case Nothing => e.Error("echo without ask");
67                 case Just(s) => e.Print(x+s);
68             }
69         case Turn(x) =>
70             if (e.turtle_exists == false) {
71                 e.Draw();
72                 e.turtle_exists := true;
73             }
74             e.Turn(bigstep_D_level1(x));
75         case Forward(x) =>
76             if (e.turtle_exists == false) {
77                 e.Draw();
78                 e.turtle_exists := true;
79             }
80             e.Forward(bigstep_P_level1(x));
81         case Seq(c1, c2) => {
82             bigstep_level1(c1, e);
83             bigstep_level1(c2, e);
84         }
85     }
86 }
```

B: Dafny implementation of Level 2

```
// Level 2

datatype Com2 = Print(string)
              | IsAsk(string, string)
              | Assign(string, string)
              | Turn(O2)
              | Forward(O2)
              | Sleep(O2)
              | Seq(Com2, Com2)

datatype O2 = Empty
            | N(int)
            | V(string)

type Mem2 = map<string, string> // Loc = string, value string,
   converteren voor int (onderaan)

trait Env2
{
    var turtle_exists: bool
    var sigma: Mem2 // memory var-val
    method Print(output: string) modifies this
    method Ask(question: string) returns (input: string) modifies
       this
    method Turn(angle: int) modifies this
    method Forward(step: int) modifies this
    method Draw() modifies this
    method Assign(variable: string, value: string) modifies this
    {
        sigma := sigma[variable := value];
    }
    method Sleep(time: int) modifies this
    method SubstText(plaintext: string) returns (valuestext: string)
       modifies this
    {
        var indexbegin := |plaintext| + 1;
        var indexend := |plaintext| + 1;
        for i := 0 to |plaintext| {
            for j := i to |plaintext| {
                if (plaintext[i..j] in sigma){
                    indexbegin := i;
                    indexend := j;
                    valuestext := valuestext + sigma[plaintext[i..j
                       ]];
                }
            }
            if (i < indexbegin || i > indexend){ // now we can be
               sure that there is no var at index i
```

```
44                  valuestext := valuestext + [plaintext[i]];
45              }
46          }
47      }
48      method SubstNumber(name: string) returns (number: int) modifies
           this
49      {
50          if(name in sigma && |sigma[name]| > 0)
51          {
52              var temp := strToInt(sigma[name]);
53              match(temp){
54                  case Just(n) => number := n;
55                  case Nothing => Error("not a number");
56              }
57          }
58          else {
59              Error("not a variable");
60          }
61      }
62      method Error(exception: string) modifies this // turn,forward,
           sleep used with string instead of int
63  }
64
65  type Config2 = (Com2, Env2)
66  type Derivation2 = seq<Config2>
67
68  method bigstep_level2(c: Com2, e: Env2)
69      modifies e
70  {
71      match(c){
72          case Print(x) =>
73              var temp := e.SubstText(x);
74              e.Print(temp);
75          case IsAsk(x, y) =>
76              var answer := e.Ask(y); // questions cannot yet have
                   variables, so no subst call
77              e.Assign(x, answer);
78          case Assign(x, y) =>
79              e.Assign(x,y);
80          case Turn(x) =>
81              var num : int;
82              match(x) {
83                  case Empty => num := -90;
84                  case N(n) => num := n;
85                  case V(n) => num := e.SubstNumber(n);
86              }
87              if (e.turtle_exists == false) {
88                  e.Draw();
89                  e.turtle_exists := true;
90              }
```

```
91              e.Turn(num);
92          case Forward(x) =>
93              var num : int;
94              match(x) {
95                  case Empty => num := 50;
96                  case N(n) => num := n;
97                  case V(n) => num := e.SubstNumber(n);
98              }
99              if (e.turtle_exists == false) {
100                 e.Draw();
101                 e.turtle_exists := true;
102             }
103             e.Forward(num);
104         case Sleep(x) =>
105             var num : int;
106             match(x) {
107                 case Empty => num := 1;
108                 case N(n) => num := n;
109                 case V(n) => num := e.SubstNumber(n);
110             }
111             e.Sleep(num);
112         case Seq(c1, c2) =>
113             bigstep_level2(c1, e);
114             bigstep_level2(c2, e);
115     }
116 }
117
118 // everything from here is for string to int conversion
119
120 datatype Maybe<T> = Nothing | Just(T)
121
122 predicate method isInt(a: char)
123 {
124   a as int - '0' as int <= 9
125 }
126
127
128 function method charToInt(a : char): int
129 requires isInt(a)
130 {
131   a as int - '0' as int
132 }
133
134 method strToInt(a : string) returns(r : Maybe<int>)
135   requires |a| > 0
136   ensures (forall k :: 0 <= k < |a| ==> isInt(a[k])) ==> exists x ::
        r == Just(x)
137   ensures (exists k :: 0 <= k < |a| && !isInt(a[k])) ==> r == Nothing
138 {
139   if (isInt(a[0])){
```

```
140       r := Just (charToInt(a[0]));
141     }
142     else {
143       r := Nothing;
144     }
145
146     assert !isInt(a[0]) ==> r == Nothing;
147
148     for j := 1 to |a|
149       invariant (forall k :: 0 <= k < |a| ==> isInt(a[k])) ==> exists x
              :: r == Just(x)
150       invariant forall k :: 0 <= k < j ==> !isInt(a[k]) ==> r ==
             Nothing
151     {
152       match r {
153         case Nothing => r := Nothing;
154         case Just(x) =>
155           if (isInt(a[j])){
156             r := Just (x * 10 + charToInt(a[j]));
157           }
158           else {
159             r := Nothing;
160           }
161       }
162     }
163 }
```

C: Dafny implementation of Level 3

```
// Level 3 and 4
datatype Com3 = Print(string)
              | IsAsk(string, string)
              | Assign(string, M)
              | Turn(O3)
              | Forward(O3)
              | Sleep(O3)
              | AddTo(M, L)
              | RemoveFrom(M, L)
              | Seq(Com3, Com3)

datatype O3 = Empty
            | N(int)
            | V(string)
            | AtRandom(L)

datatype M = N(int) | V(string) | L(seq<M>) // list not uniform

type L = seq<M>

type Mem3 = map<string, M> // Loc = string

trait Env3
{
    var turtle_exists: bool
    var sigma: Mem3 // memory var-val
    method Print(output: string) modifies this
    method Ask(question: string) returns (input: M) modifies this
    method Turn(angle: int) modifies this
    method Forward(step: int) modifies this
    method Draw() modifies this
    method Assign(variable: string, value: M) modifies this
    {
        sigma := sigma[variable := value];
    }
    method Sleep(time: int) modifies this
    method SubstText(plaintext: string) returns (valuestext: string)
       modifies this
    {
        var indexbegin := |plaintext| + 1;
        var indexend := |plaintext| + 1;
        for i := 0 to |plaintext| {
            for j := i to |plaintext| {
                if (plaintext[i..j] in sigma){
                    indexbegin := i;
                    indexend := j;
                    match (sigma[plaintext[i..j]])
                        case N(x) =>
```

```
48                              var temp := intToStr(x);
49                              valuestext := valuestext + temp;
50                          case V(x) =>
51                              valuestext := valuestext + x;
52                          case L(x) =>
53                              Error("Cannot print a list");
54                      }
55                  }
56                  if (i < indexbegin || i > indexend){ // now we can be
                        sure that there is no var at index i
57                      valuestext := valuestext + [plaintext[i]];
58                  }
59              }
60          }
61          method SubstNumber(name: string) returns (number: int) modifies
                this
62          {
63              if(name in sigma)
64              {
65                  match (sigma[name]){
66                      case N(x) => number := x;
67                      case V(x) => Error("the value of this variable is not
                            a number but a string");
68                      case L(x) => Error("the value of this variable is not
                            a number but a list");
69                  }
70              }
71              else {
72                  Error("not a variable");
73              }
74          }
75          method Error(exception: string) modifies this
76          method AddTo(element: M, list: L) modifies this
77          method RemoveFrom(element: M, list: L) modifies this
78          method Random(list: L) returns (element: M) modifies this
79      }
80
81      type Config3 = (Com3, Env3)
82      type Derivation3 = seq<Config3>
83
84      method bigstep_level3(c: Com3, e: Env3)
85          modifies e
86      {
87          match(c){
88              case Print(x) =>
89                  var temp := e.SubstText(x);
90                  e.Print(temp);
91              case IsAsk(x, y) =>
92                  var answer := e.Ask(y); // questions cannot yet have
                        variables, so no subst call
```

```
93              e.Assign(x, answer);
94          case Assign(x, y) =>
95              e.Assign(x, y);
96          case Turn(x) =>
97              var num : int;
98              match(x) { //x is of type O3
99                  case Empty => num := -90;
100                 case N(n) => num := n;
101                 case V(n) => num := e.SubstNumber(n);
102                 case AtRandom(l) => {
103                     var temp := e.Random(l);
104                     match(temp){ // temp is of type M
105                         case N(m) => num := m;
106                         case V(m) => e.Error("Using variables through
                                a list and at random is not accepted");
107                         case L(m) => e.Error("Nesting lists, so using
                                lists of lists is not accepted");
108                     }
109                 }
110             }
111             if (e.turtle_exists == false) {
112                 e.Draw();
113                 e.turtle_exists := true;
114             }
115             e.Turn(num);
116         case Forward(x) =>
117             var num : int;
118             match(x) {
119                 case Empty => num := 50;
120                 case N(n) => num := n;
121                 case V(n) => num := e.SubstNumber(n);
122                 case AtRandom(l) => {
123                     var temp := e.Random(l);
124                     match(temp){ // temp is of type M
125                         case N(m) => num := m;
126                         case V(m) => e.Error("Using variables through
                                a list and at random is not accepted");
127                         case L(m) => e.Error("Nesting lists, so using
                                lists of lists is not accepted");
128                     }
129                 }
130             }
131             if (e.turtle_exists == false) {
132                 e.Draw();
133                 e.turtle_exists := true;
134             }
135             e.Forward(num);
136         case Sleep(x) =>
137             var num : int;
138             match(x) {
```

```
139                     case Empty => num := 1;
140                     case N(n) => num := n;
141                     case V(n) => num := e.SubstNumber(n);
142                     case AtRandom(l) => {
143                         var temp := e.Random(l);
144                         match(temp){ // temp is of type M
145                             case N(m) => num := m;
146                             case V(m) => e.Error("Using variables through
                                    a list and at random is not accepted");
147                             case L(m) => e.Error("Nesting lists, so using
                                    lists of lists is not accepted");
148                         }
149                     }
150                 }
151                 e.Sleep(num);
152             case AddTo(x, l) =>
153                 e.AddTo(x,l);
154             case RemoveFrom(x, l) =>
155                 e.RemoveFrom(x,l);
156             case Seq(c1, c2) =>
157                 bigstep_level3(c1, e);
158                 bigstep_level3(c2, e);
159         }
160 }
161
162 // everything from here is for conversion string to int or int to
       string
163
164 datatype Maybe<T> = Nothing | Just(T)
165
166 predicate method isInt(a: char)
167 {
168   a as int - '0' as int <= 9
169 }
170
171 function method charToInt(a : char): int
172 requires isInt(a)
173 {
174   a as int - '0' as int
175 }
176
177 function method intToChar(a : nat): char
178 requires a <= 9
179 {
180   a as char + '0' as char
181 }
182
183 method strToInt(a : string) returns(r : Maybe<int>)
184   requires |a| > 0
185   ensures (forall k :: 0 <= k < |a| ==> isInt(a[k])) ==> exists x ::
```

```
      r == Just(x)
  ensures (exists k :: 0 <= k < |a| && !isInt(a[k])) ==> r == Nothing
{
  if (isInt(a[0])){
    r := Just (charToInt(a[0]));
  }
  else {
    r := Nothing;
  }

  assert !isInt(a[0]) ==> r == Nothing;

  for j := 1 to |a|
    invariant (forall k :: 0 <= k < |a| ==> isInt(a[k])) ==> exists x
        :: r == Just(x)
    invariant forall k :: 0 <= k < j ==> !isInt(a[k]) ==> r ==
       Nothing
  {
    match r {
      case Nothing => r := Nothing;
      case Just(x) =>
        if (isInt(a[j])){
          r := Just (x * 10 + charToInt(a[j]));
        }
        else {
          r := Nothing;
        }
    }
  }
}

method intToStr(a : int) returns(r : string)
{
  r := "";
  var temp1 := a;
  while temp1 > 0
  decreases temp1
  {
    var temp2 := temp1 % 10;
    r := [intToChar(temp2)] + r;
    temp1 := (temp1 - temp2) / 10;
  }
}
```

D: Dafny implementation of Level 4

```
// Level 3 and 4
datatype Com3 = Print(string)
              | IsAsk(string, string)
              | Assign(string, M)
              | Turn(O3)
              | Forward(O3)
              | Sleep(O3)
              | AddTo(M, L)
              | RemoveFrom(M, L)
              | Seq(Com3, Com3)

datatype O3 = Empty
            | N(int)
            | V(string)
            | AtRandom(L)

datatype M = N(int) | V(string) | L(seq<M>) // list not uniform

type L = seq<M>

type Mem3 = map<string, M> // Loc = string

trait Env3
{
    var turtle_exists: bool
    var sigma: Mem3 // memory var-val
    method Print(output: string) modifies this
    method Ask(question: string) returns (input: M) modifies this
    method Turn(angle: int) modifies this
    method Forward(step: int) modifies this
    method Draw() modifies this
    method Assign(variable: string, value: M) modifies this
    {
        sigma := sigma[variable := value];
    }
    method Sleep(time: int) modifies this
    method SubstText(plaintext: string) returns (valuestext: string)
       modifies this
    {
        var indexbegin := |plaintext| + 1;
        var indexend := |plaintext| + 1;
        var quotes := false;
        for i := 0 to |plaintext| {
            if (plaintext[i] == '\''){
                quotes := !quotes;
            }
            for j := i to |plaintext| {
                if (plaintext[i..j] in sigma && !quotes){
```

```
48                      indexbegin := i;
49                      indexend := j;
50                      match (sigma[plaintext[i..j]])
51                          case N(x) =>
52                              Error("Text used without quotes");
53                          case V(x) =>
54                              valuestext := valuestext + x;
55                          case L(x) =>
56                              Error("Cannot print a list");
57                  }
58              }
59              if ((i < indexbegin || i > indexend) && plaintext[i] != '
                    \'' ){ // now we can be sure that there is no var at
                    index i
60                  valuestext := valuestext + [plaintext[i]];
61              }
62          }
63      }
64      method SubstNumber(name: string) returns (number: int) modifies
            this
65      {
66          if(name in sigma)
67          {
68              match (sigma[name]){
69                  case N(x) => number := x;
70                  case V(x) => Error("the value of this variable is not
                        a number but a string");
71                  case L(x) => Error("the value of this variable is not
                        a number but a list");
72              }
73          }
74          else {
75              Error("not a variable");
76          }
77      }
78      method Error(exception: string) modifies this
79      method AddTo(element: M, list: L) modifies this
80      method RemoveFrom(element: M, list: L) modifies this
81      method Random(list: L) returns (element: M) modifies this
82  }
83
84  type Config3 = (Com3, Env3)
85  type Derivation3 = seq<Config3>
86
87  method bigstep_level3(c: Com3, e: Env3)
88      modifies e
89  {
90      match(c){
91          case Print(x) =>
92              var temp := e.SubstText(x);
```

```
93              e.Print(temp);
94          case IsAsk(x, y) =>
95              var answer := e.Ask(y); // questions cannot yet have
                    variables, so no subst call
96              e.Assign(x, answer);
97          case Assign(x, y) =>
98              e.Assign(x, y);
99          case Turn(x) =>
100             var num : int;
101             match(x) { //x is of type O3
102                 case Empty => num := -90;
103                 case N(n) => num := n;
104                 case V(n) => num := e.SubstNumber(n);
105                 case AtRandom(l) => {
106                     var temp := e.Random(l);
107                     match(temp){ // temp is of type M
108                         case N(m) => num := m;
109                         case V(m) => e.Error("Using variables through
                                a list and at random is not accepted");
110                         case L(m) => e.Error("Nesting lists, so using
                                lists of lists is not accepted");
111                     }
112                 }
113             }
114             if (e.turtle_exists == false) {
115                 e.Draw();
116                 e.turtle_exists := true;
117             }
118             e.Turn(num);
119         case Forward(x) =>
120             var num : int;
121             match(x) {
122                 case Empty => num := 50;
123                 case N(n) => num := n;
124                 case V(n) => num := e.SubstNumber(n);
125                 case AtRandom(l) => {
126                     var temp := e.Random(l);
127                     match(temp){ // temp is of type M
128                         case N(m) => num := m;
129                         case V(m) => e.Error("Using variables through
                                a list and at random is not accepted");
130                         case L(m) => e.Error("Nesting lists, so using
                                lists of lists is not accepted");
131                     }
132                 }
133             }
134             if (e.turtle_exists == false) {
135                 e.Draw();
136                 e.turtle_exists := true;
137             }
```

```
138             e.Forward(num);
139         case Sleep(x) =>
140             var num : int;
141             match(x) {
142                 case Empty => num := 1;
143                 case N(n) => num := n;
144                 case V(n) => num := e.SubstNumber(n);
145                 case AtRandom(l) => {
146                     var temp := e.Random(l);
147                     match(temp){ // temp is of type M
148                         case N(m) => num := m;
149                         case V(m) => e.Error("Using variables through
                                a list and at random is not accepted");
150                         case L(m) => e.Error("Nesting lists, so using
                                lists of lists is not accepted");
151                     }
152                 }
153             }
154             e.Sleep(num);
155         case AddTo(x, l) =>
156             e.AddTo(x,l);
157         case RemoveFrom(x, l) =>
158             e.RemoveFrom(x,l);
159         case Seq(c1, c2) =>
160             bigstep_level3(c1, e);
161             bigstep_level3(c2, e);
162     }
163 }
164
165 // everything from here is for conversion string to int or int to
       string
166
167 datatype Maybe<T> = Nothing | Just(T)
168
169 predicate method isInt(a: char)
170 {
171   a as int - '0' as int <= 9
172 }
173
174 function method charToInt(a : char): int
175 requires isInt(a)
176 {
177   a as int - '0' as int
178 }
179
180 function method intToChar(a : nat): char
181 requires a <= 9
182 {
183   a as char + '0' as char
184 }
```

```
185
186 method strToInt(a : string) returns(r : Maybe<int>)
187   requires |a| > 0
188   ensures (forall k :: 0 <= k < |a| ==> isInt(a[k])) ==> exists x ::
        r == Just(x)
189   ensures (exists k :: 0 <= k < |a| && !isInt(a[k])) ==> r == Nothing
190 {
191   if (isInt(a[0])){
192     r := Just (charToInt(a[0]));
193   }
194   else {
195     r := Nothing;
196   }
197
198   assert !isInt(a[0]) ==> r == Nothing;
199
200   for j := 1 to |a|
201     invariant (forall k :: 0 <= k < |a| ==> isInt(a[k])) ==> exists x
          :: r == Just(x)
202     invariant forall k :: 0 <= k < j ==> !isInt(a[k]) ==> r ==
          Nothing
203   {
204     match r {
205       case Nothing => r := Nothing;
206       case Just(x) =>
207         if (isInt(a[j])){
208           r := Just (x * 10 + charToInt(a[j]));
209         }
210         else {
211           r := Nothing;
212         }
213     }
214   }
215 }
216
217 method intToStr(a : int) returns(r : string)
218 {
219   r := "";
220   var temp1 := a;
221   while temp1 > 0
222   decreases temp1
223   {
224     var temp2 := temp1 % 10;
225     r := [intToChar(temp2)] + r;
226     temp1 := (temp1 - temp2) / 10;
227   }
228 }
```

E: Dafny implementation of Level 5

```
// Level 5

datatype Com5 = Print(string)
              | IsAsk(string, string)
              | Assign(string, M)
              | Turn(O5)
              | Forward(O5)
              | Sleep(O5)
              | AddTo(M, L)
              | RemoveFrom(M, L)
              | If(B, Com5)
              | IfElse(B, Com5, Com5)
              | Seq(Com5, Com5)

datatype O5 = Empty | N(int) | V(string) | AtRandom(L) //O5 same as
   O3, still used from level 3

datatype M = N(int) | V(string) | L(seq<M>) //from level 3 still used

type L = seq<M> // still used from level 3

datatype B = Eq(M, M) // M is M
           | Elem(M, L) // string in L

type Mem5 = map<string, M> // Loc = string

trait Env5
{
    var turtle_exists: bool
    var sigma: Mem5 // memory var-val
    method Print(output: string) modifies this
    method Ask(question: string) returns (input: M) modifies this
    method Turn(angle: int) modifies this
    method Forward(step: int) modifies this
    method Draw() modifies this
    method Assign(variable: string, value: M) modifies this
    {
        sigma := sigma[variable := value];
    }
    method Sleep(time: int) modifies this
    method SubstText(plaintext: string) returns (valuestext: string)
       modifies this
    {
        var indexbegin := |plaintext| + 1;
        var indexend := |plaintext| + 1;
        var quotes := false;
        for i := 0 to |plaintext| {
            if (plaintext[i] == '\''){
```

```
47                  quotes := !quotes;
48              }
49              for j := i to |plaintext| {
50                  if (plaintext[i..j] in sigma && !quotes){
51                      indexbegin := i;
52                      indexend := j;
53                      match (sigma[plaintext[i..j]])
54                          case N(x) =>
55                              Error("Text used without quotes");
56                          case V(x) =>
57                              valuestext := valuestext + x;
58                          case L(x) =>
59                              Error("Cannot print a list");
60                  }
61              }
62              if ((i < indexbegin || i > indexend) && plaintext[i] != '
                    \''){ // now we can be sure that there is no var at
                    index i
63                  valuestext := valuestext + [plaintext[i]];
64              }
65          }
66      }
67      method SubstNumber(name: string) returns (number: int) modifies
            this
68      {
69          if(name in sigma)
70          {
71              match (sigma[name]){
72                  case N(x) => number := x;
73                  case V(x) => Error("the value of this variable is not
                        a number but a string");
74                  case L(x) => Error("the value of this variable is not
                        a number but a list");
75              }
76          }
77          else {
78              Error("not a variable");
79          }
80      }
81      method Error(exception: string) modifies this
82      method AddTo(element: M, list: L) modifies this
83      method RemoveFrom(element: M, list: L) modifies this
84      method Random(list: L) returns (element: M) modifies this
85      method Eq(x: M, y: M) returns (cond: bool) modifies this
86      {
87          match(x, y) {
88              case (N(n1), N(n2)) =>
89                  cond := n1 == n2;
90              case (N(n), V(v)) =>
91                  var temp := SubstNumber(v);
```

```
92                  cond := n == temp;
93              case (N(n), L(l)) =>
94                  Error("Cannot compare list with int");
95              case (V(v), N(n)) =>
96                  var temp := SubstNumber(v);
97                  cond := n == temp;
98              case (V(v1), V(v2)) =>
99                  var temp1 := SubstText(v1);
100                 var temp2 := SubstText(v2);
101                 cond := temp1 == temp2;
102             case (V(v), L(l)) =>
103                 Error("Cannot compare variable or string with list");
104             case (L(l), N(n)) =>
105                 Error("Cannot compare list with int");
106             case (L(l), V(v)) =>
107                 Error("Cannot compare variable or string with list");
108             case (L(l1), L(l2)) =>
109                 cond := l1 == l2;
110         }
111     }
112     method Elem(x: M, y: L) returns (cond: bool) modifies this
113     {
114         cond := false;
115         for i := 0 to |y| {
116             if (x == y[i] && cond == false) {cond := true;}
117         }
118     }
119 }
120
121 type Config5 = (Com5, Env5)
122 type Derivation5 = seq<Config5>
123
124
125
126 method bigstep_level5(c: Com5, e: Env5)
127     modifies e
128 {
129     match(c){
130         case Print(x) =>
131             var temp := e.SubstText(x);
132             e.Print(temp);
133         case IsAsk(x, y) =>
134             var answer := e.Ask(y); // questions cannot yet have
                    variables, so no subst call
135             e.Assign(x, answer);
136         case Assign(x, y) =>
137             e.Assign(x, y);
138         case Turn(x) =>
139             var num : int;
140             match(x) { //x is of type O3
```

```
141                 case Empty => num := -90;
142                 case N(n) => num := n;
143                 case V(n) => num := e.SubstNumber(n);
144                 case AtRandom(l) => {
145                     var temp := e.Random(l);
146                     match(temp){ // temp is of type M
147                         case N(m) => num := m;
148                         case V(m) => e.Error("Using variables through
                                a list and at random is not accepted");
149                         case L(m) => e.Error("Nesting lists, so using
                                lists of lists is not accepted");
150                     }
151                 }
152             }
153             if (e.turtle_exists == false) {
154                 e.Draw();
155                 e.turtle_exists := true;
156             }
157             e.Turn(num);
158         case Forward(x) =>
159             var num : int;
160             match(x) {
161                 case Empty => num := 50;
162                 case N(n) => num := n;
163                 case V(n) => num := e.SubstNumber(n);
164                 case AtRandom(l) => {
165                     var temp := e.Random(l);
166                     match(temp){ // temp is of type M
167                         case N(m) => num := m;
168                         case V(m) => e.Error("Using variables through
                                a list and at random is not accepted");
169                         case L(m) => e.Error("Nesting lists, so using
                                lists of lists is not accepted");
170                     }
171                 }
172             }
173             if (e.turtle_exists == false) {
174                 e.Draw();
175                 e.turtle_exists := true;
176             }
177             e.Forward(num);
178         case Sleep(x) =>
179             var num : int;
180             match(x) {
181                 case Empty => num := 1;
182                 case N(n) => num := n;
183                 case V(n) => num := e.SubstNumber(n);
184                 case AtRandom(l) => {
185                     var temp := e.Random(l);
186                     match(temp){ // temp is of type M
```

```
187                         case N(m) => num := m;
188                         case V(m) => e.Error("Using variables through
                                a list and at random is not accepted");
189                         case L(m) => e.Error("Nesting lists, so using
                                lists of lists is not accepted");
190                     }
191                 }
192             }
193             e.Sleep(num);
194         case AddTo(x, l) =>
195             e.AddTo(x,l);
196         case RemoveFrom(x, l) =>
197             e.RemoveFrom(x,l);
198         case If(b, c1) =>
199             var condition : bool;
200             match(b) {
201                 case Eq(x, y) => condition := e.Eq(x, y);
202                 case Elem(x, y) => condition := e.Elem(x, y);
203             }
204             if (condition) {bigstep_level5(c1, e);}
205         case IfElse(b, c1, c2) =>
206             var condition : bool;
207             match(b) {
208                 case Eq(x, y) => condition := e.Eq(x, y);
209                 case Elem(x, y) => condition := e.Elem(x, y);
210             }
211             if (condition) {bigstep_level5(c1, e);} else {
                    bigstep_level5(c2, e);}
212         case Seq(c1, c2) =>
213             bigstep_level5(c1, e);
214             bigstep_level5(c2, e);
215     }
216 }
217
218 // everything from here is for conversion of string to int or int to
    string
219
220 datatype Maybe<T> = Nothing | Just(T)
221
222 predicate method isInt(a: char)
223 {
224   a as int - '0' as int <= 9
225 }
226
227 function method charToInt(a : char): int
228 requires isInt(a)
229 {
230   a as int - '0' as int
231 }
232
```

```
233 function method intToChar(a : nat): char
234 requires a <= 9
235 {
236   a as char + '0' as char
237 }
238
239 method strToInt(a : string) returns(r : Maybe<int>)
240   requires |a| > 0
241   ensures (forall k :: 0 <= k < |a| ==> isInt(a[k])) ==> exists x ::
        r == Just(x)
242   ensures (exists k :: 0 <= k < |a| && !isInt(a[k])) ==> r == Nothing
243 {
244   if (isInt(a[0])){
245     r := Just (charToInt(a[0]));
246   }
247   else {
248     r := Nothing;
249   }
250
251   assert !isInt(a[0]) ==> r == Nothing;
252
253   for j := 1 to |a|
254     invariant (forall k :: 0 <= k < |a| ==> isInt(a[k])) ==> exists x
          :: r == Just(x)
255     invariant forall k :: 0 <= k < j ==> !isInt(a[k]) ==> r ==
         Nothing
256   {
257     match r {
258       case Nothing => r := Nothing;
259       case Just(x) =>
260         if (isInt(a[j])){
261           r := Just (x * 10 + charToInt(a[j]));
262         }
263         else {
264           r := Nothing;
265         }
266     }
267   }
268 }
269
270 method intToStr(a : int) returns(r : string)
271 {
272   r := "";
273   var temp1 := a;
274   while temp1 > 0
275   decreases temp1
276   {
277     var temp2 := temp1 % 10;
278     r := [intToChar(temp2)] + r;
279     temp1 := (temp1 - temp2) / 10;
```

```
280         }
281     }
```